Universität Dortmund
Fachbereich Informatik

Projektgruppe 425
Lernende, autonome Fußballroboter
Endbericht

M. Arbatzat, S. Freitag, M. Fricke, C. Heermann,K. Hegelich,
A. Krause, J. Krüger, H. Müller, J. Schanko, M. Schulte-Hobein, M. Theile, S. Welker

24. September 2003

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Allgemeines

Im Rahmen der Projektgruppe 425 „Lernende autonome Fußballroboter“, die im Wintersemester 2002/03 und Sommersemester 2003 stattfand, sollte ein Roboter in die Lage versetzt werden, Fußball zu spielen. Aus Sicht der Informatik und speziell der Künstlichen Intelligenz ergaben sich für uns als zentrale Probleme: die Bildverarbeitung (Vision) als unser primärer Sensor, das Erstellen eines internen Weltbildes, das Planen und Ausführen von Aufgaben (Navigationsplanung) und die Manipulation von realen Objekten, z.B. die Bewegungsänderung des Fußballs. Für die Konstruktion eines Fußballroboters sind ebenso elektrotechnische und mechanische Ansätze von Bedeutung.

Unser Ziel war der Entwurf eines ausfallsicheren, fußballspielenden Roboters mit einer robusten und stabilen Bildverarbeitung. Wir haben uns früh entschieden, nicht nur einen Fußballroboter zu konstruieren und zu programmieren, sondern ein ganzes Team, das an den German Open 2003 in Paderborn teilnehmen sollte, auf die Beine zu stellen. Zur Robocup-Weltmeisterschaft 2003 in Padua (Italien) haben wir unsere Erfahrungen aus den German Open genutzt, um in für uns wichtigen Teilaufgaben wie Schussgeschwindigkeit, Selbstlokalisation, Pfadplanung und Fahrverhalten eine Optimierung zu erzielen.

Zu Beginn wurden uns zur Verfügung gestellt: ein Fahrwerk, das man über einen Motorcontroller ansteuern konnte und eine Kamera, die man ebenfalls mittels einer einsatzfähigen Schnittstelle nutzen konnte. Diese beiden Grundkomponenten sollten effektiv eingesetzt werden. Dazu war es entscheidend, sich mit der Ansteuerung der Motoren und dem dafür verwendeten Protokoll auseinanderzusetzen, sowie mit der Elektronik des Motorcontrollers, über die ein Ballschuß ausgelöst werden kann. Die Kamera wird als der zentrale Sensor angesehen und zur Erkennung von Objekten eingesetzt. Die Bildverarbeitung muss dabei robust und schnell arbeiten, weil sich die Umgebung, in der die Fußballroboter agieren, sehr schnell ändert. Eine zuverlässige Bildverarbeitung ist notwendig, um die Sensorinformationen zur Selbstlokalisation und damit zur Erstellung und Aufrechterhaltung eines Weltbildes verwenden zu können. Welche Aktion zu einem Zeitpunkt gewählt wird, hängt von einem aktuellen Weltbild ab. Wie diese Entscheidung aber getroffen wird, gehörte ebenso zu den zu lösenden Aufgaben der Projektgruppe. Neben Einschränkungen, die weiter unten erläutert werden, mussten bei der Konstruktion noch weitere Aspekte berücksichtigt werden. Dazu zählen beispielsweise das Gewicht und die Ausmaße des zur Verfügung gestellten Fahrwerks, da man dieses weiter nutzen wollte.

1.2 Robocup

Die Robocup-Initiative[1] wurde 1997 gegründet, sie dient der Forschung und Lehre innerhalb der Robotik und der Künstlichen Intelligenz. An einem Standardproblem, dem Fußballspiel, können ver-

[1] www.robocup.org

schiedene Ansätze evaluiert werden. Konferenzen, Schulungen und der einmal pro Jahr stattfindende Wettbewerb, bei dem die Forschungsgruppen ihre eigenen Entwicklungen unter dem Aspekt der Konkurrenzfähigkeit testen, sind Bestandteil der Initiative. Die Wettbewerbe sind in Ligen unterteilt. Zu der Kategorie Robocup Soccer gehören die Simulation League, Small Size Robot League (f-180), Middle Size Robot League (f-2000), Sony Legged Robot League (unterstützt von Sony) und Humanoid League (seit 2002). Neben dem Robocup Soccer-Wettbewerben finden noch die Robocup Rescue und Robocup Junior Wettkämpfe statt. Aus den unterschiedlichen Regeln und technischen Anforderungen in den einzelnen Ligen, ergeben sich verschiedene Problemstellungen und Forschungsschwerpunkte.

Unser Team nahm an den Spielen der Middle Size Robot League teil. Dort besteht eine Mannschaft aus vier Spielern inklusive dem Torwart. Die Ausmaße der Spieler sind begrenzt: ihre Höhe muss zwischen 30 cm und 85 cm liegen, und die Grundfläche darf ein Quadrat mit einer Seitenlänge von 50 cm nicht überschreiten. Externe Sensoren wie eine globale Kamera sind nicht zugelassen, die Spieler sind vollständig auf eigene Sensoren und Kameras angewiesen.

Das Spielfeld und die Spieler sind farblich so markiert, dass eine klare Unterteilung zu erkennen ist. Die Spieler müssen größtenteils schwarz sein und eine Banderole tragen. Diese Banderole trägt eine Zahl und eine spezifische Farbe: violett für die eine, türkis für die andere Mannschaft. Gespielt wird auf einem grünen, 9 m * 5 m großen Feld mit einem orangefarbenen Ball, der den FIFA-Regeln entspricht. Das Spielfeld ist nur von einer ca. 10 cm hohen Bande[2] umgeben, womit ein weiterer Faktor hinzukommt, der z.B. bei der Bildverarbeitung oder der Selbstlokalisation berücksichtigt werden muss. Die Markierungen auf dem Feld sind weiß und entsprechen denen eines gewöhnlichem Fußballfeldes. Die Tore sind 2 m breit und 1 m hoch sowie farblich von einander getrennt: das eine ist blau und das andere gelb jeweils mit weißen Pfosten. In den Spielfeldecken stehen Eckfahnen mit einer Höhe von 1 m und einem Durchmesser von ca. 30 cm. Die Eckfahnen des blauen Tors sind blau-gelb-blau gefärbt, die Eckfahnen des gelben Tores entsprechend gelb-blau-gelb. Ein Spiel dauert 2 * 10 Minuten. Die Spieler sind soweit autonom, dass sie nur beim Anpfiff und bei regelkonformen Unterbrechungen gesteuert werden dürfen, ansonsten besteht während des Spiels keine Interaktionsmöglichkeit mit Menschen. Die Spieler können jedoch untereinander Informationen austauschen. Ist ein Spieler ausgefallen, oder soll er ausgewechselt werden, kann dieser nach Absprache mit dem Schiedsrichter vom Platz genommen werden.

1.3 Projektverlauf

Die Arbeit der Projektgruppe gliederte sich in drei Phasen. In der Orientierungsphase, die vom Anfang des Wintersemesters 2002/03 bis Ende Dezember 2002 dauerte, bildeten wir drei Teams mit jeweils vier Student(en)/innen. Ziel dieser Phase war es, sich mit der Arbeitsumgebung vertraut zu machen. Dazu gehörten der Einsatz der Bibliothek `cvtk`[3][7], die Ansteuerung der Kamera und die vorhandene Schnittstelle zur Kommunikation mit dem Motorcontroller. Mit dem Ende dieser Phase musste jedes Team vorgegebene Benchmarks absolvieren. Insgesamt wurden fünf Tests durchgeführt: diese bestanden darin, aus unterschiedlichen Positionen sowie Ausrichtungen des Roboters einen orangefarbenen Ball in möglichst kurzer Zeit in ein blaues Tor zu schießen. Bei einem der Tests wurde zusätzlich der Ball bewegt, was sich als schwierigste Aufgabe erwies und nur von einer Gruppe gelöst wurde. Am Ende dieser Phase stand fest, dass wir eine omnidirektionale Kamera verwenden und ein Team für die German Open in Paderborn aufstellen würden.

Der zweite Teil der Projektarbeit wurde genutzt, um Lösungen für die erkannten Probleme der ersten Phase zu finden und sich somit auf die German Open vorzubereiten. Anhand der Erfahrungen aus der 1. Phase und den Vorträgen aus der Seminarphase ergaben sich für uns die folgenden Schwerpunkte: Bildverarbeitung, Weltmodell, Strategie, Kommunikation, Roboteransteuerung und

[2] Seit 2003 wird bei Spielen der Midsize-League keine Bande mehr verwendet.

[3] colour vision toolkit

Roboterhardware. Für jeden dieser Bereiche gab es einen Verantwortlichen, der sowohl die Rolle des Ansprechpartners als auch des Projektmanager für seine Gruppe übernahm. Es wurde ein Konzept entworfen, das thread-orientiert ausgelegt war - dieses wurde nach den German Open aufgegeben. Wurde in der ersten Phase noch auf einem kleinen 2 m * 1 m Feld getestet, benutzten wir jetzt einen Simulator (vgl. Abschnitt 3.7.2) und ein Spielfeld, das sich an den Regeln der Midsize-League[1] orientierte, um unsere Arbeit zu validieren. Das Testfeld war grün mit weißen Markern für die Tor- und Seitenauslinien. Die oben genannten Marker wurden zur Verifikation einiger Tests eingesetzt. Das Feld maß ca. 8 m in der Länge und ca. 4 m in der Breite. An den Ecken befanden sich 1 m hohe Eckfahnen, die auf der Seite des blauen Tors blau-gelb-blau gestrichen waren und gelb-blau-gelb auf Seiten des gelben Tors. Ziel dieser Phase war es, am Ende eine stabile Bildverarbeitung und einen insgesamt robusten Fußballroboter zu haben, mit dem wir an den German Open 2003 in Paderborn teilnehmen konnten.

In der letzten Phase wurden Optimierungen am Programm und der Roboter-Hardware durchgeführt, um eine erfolgreiche Teilnahme bei der Weltmeisterschaft 2003 in Padua (Italien) zu garantieren. Für die WM wurde aufgrund der einfacheren Kontrollierbarkeit das Thread-Konzept durch ein serielles ersetzt, nähere Informationen liefert Abschnitt 3.1.2. Weitere Verbesserungen wurden in den Bereichen Pfadplanung, Selbstlokalisation, Bildverarbeitung, Ansteuerung der Motoren, Kommunikation und Torwart angestrebt, da sich bei den German Open u. a. gezeigt hatte, dass sich die Roboter noch nicht schnell genug auf dem Spielfeld bewegten, dass in manchen Situationen der Verteidiger zurück in die eigene Hälfte fuhr, obwohl das gegnerische Tor leer stand, oder dass dieser wegen der fehlenden Schusseinheit kein Tor erzielte. Jeder Roboter erhielt konsequenterweise eine Schusseinheit. Das Design des Torwarts wurde verändert: der Kicker des Torwarts wurde im Verhältnis zu den anderen Spielern breiter und an den Seiten wurden Verstrebungen angebracht, so dass der Torwart mehr Raum im Tor einnahm.

1.4 Übersicht

Das Kapitel 2 informiert über die von uns eingesetzte Hardware und das Design des Roboters. Zum Fußballspielen benötigt ein Roboter verschiedene Sensoren und Aktoren, die seine Wahrnehmungs- und Reaktionsmöglichkeiten bestimmen. Der Sensor, der bei uns am häufigsten genutzt wird, ist das Kamerasystem, wodurch wir ein 360° Abbild der Umgebung erhalten. Der Aufbau und die Funktionsweise der Kamera werden ebenso erläutert wie das omnidirektionales Fahrwerk und die Schusseinheit. Diese Komponenten werden vom Motorcontroller angesteuert. Der Motorcontroller erhält Befehle vom Steuerrechner auf dem das Hauptprogramm läuft.

In Kapitel 3 beschreiben wir die von uns entwickelte und eingesetzte Software sowie die algorithmischen Ansätze, die wir verwenden. Die Struktur des Hauptprogramms verdeutlicht Abschnitt 3.1. Um eine genauere Vorhersage bei der Fahrtplanung o. a. zu gewährleisten, entschieden wir uns für einen sequentiellen Ablauf des Programms. Die Bildverarbeitung leistet die Einteilung des aufgenommen Bildes in Farbklassen und beschränkt die Suche nach Objekten auf bestimmte Bereiche, in denen sich z.B. der Ball befinden könnte. Weitere Merkmale der Bildverarbeitung sind die Distanzfunktion und die Möglichkeit, ein Objekt zu tracken. Das Weltmodell setzt die Informationen der Bildverarbeitung zur Selbstlokalisation des Roboters auf dem Spielfeld ein. Dabei wird die wahrscheinlichste Position des Fußballroboters berechnet. Zusätzliches Wissen über die Umwelt, den Ball, die beiden Tore und die Hindernisse ist vorhanden. Die Strategie unseres Roboters ist reaktiv. Entscheidungen werden anhand von aktuellen Daten gefällt. Es wird in Grundfähigkeiten und höhere Fähigkeiten unterschieden, die erstgenannten sollen auch bei ausgefallener Selbstlokalisation funktionieren. Unsere Fußballroboter erhalten von einem zentralen Programm zusätzliche Kommandos, die von der Strategie berücksichtigt werden. Das Robotermodul kapselt alle Hardware-nahen Funktionen und stellt dem Anwender nach außen hin eine einfache Schnittstelle zur Ansteuerung der von uns eingesetzten Motorcontroller bereit. Die fremden Programme, die wir verwendet haben, beschreiben wir am Ende

dieses Kapitels.

Auf die durchgeführten Systemstests geht Kapitel 4 ein. Zum Abschluss werden wir das Erreichte darstellen und Perspektiven aufzeigen. Im Anhang findet man die Anleitung für das Hauptprogramm *Tribots* und das Remote-Programm *Tribots-Control* sowie technische Informationen zur Hardware.

Kapitel 2

Hardware

2.1 Überblick

Die erste Überlegung, die im Hinblick auf das Hardwaredesign im Rahmen der Projektgruppe angestellt wurde, betraf das zu verwendende Bildverarbeitungssystem. Zur Diskussion standen zwei grundsätzlich verschiedene Ansätze:

- eine direktionale Kamera: sie besitzt ein begrenztes Sichtfeld, in etwa vergleichbar mit dem menschlichen, so dass zu einem Zeitpunkt nur ein Bruchstück der Umgebung wahrgenommen werden kann.
- eine omnidirektionale Kamera: sie nimmt die gesamte Umgebung rund um den Roboter gleichzeitig wahr, ein 360°-Rundumblick. Realisiert wird dieses, indem die Kamera auf einen entsprechend geformten Spiegel blickt, in dem sich das gesamte Umfeld widerspiegelt.

Letzterer Ansatz bietet einen großen Vorteil, bedeutete aber die Inkaufnahme eines direkt damit verbundenen Nachteils: nicht-lineare Entfernungen im Spiegelbild aufgrund der Spiegelform. Nach Meinung der Projektgruppe überwog der Nutzen und sie entschied sich deshalb zugunsten der omnidirektionalen Lösung. Der weitere Aufbau der Roboterhardware wurde für diese Lösung konzipiert.

2.1.1 Übersicht über den Aufbau des Roboters

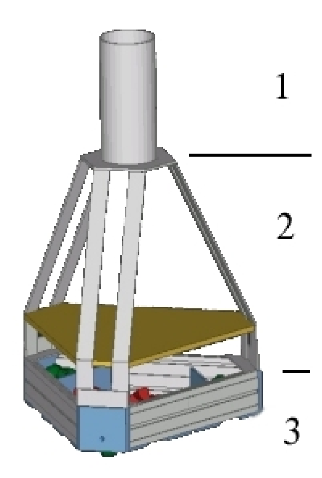


Abbildung 2.1: Roboteraufbau. In der Abbildung erkennbar sind die drei Module Fahrwerk (1), Kontroll-/Steuerebene (2), und Kameraaufbau (3).

Bei dem generellen Aufbau der Roboter lassen sich grob drei Module unterscheiden (vgl. Abbildung 2.1):

1. das Fahrwerk
2. die Kontroll- und Steuerebene
3. der Kameraaufbau mit dazugehörigem Spiegel für die omnidirektionale Sicht

Die einzelnen Module wurden so konstruiert, dass sie mit vertretbarem Aufwand voneinander getrennt werden können. Beispielsweise müssen - neben einigen Kabeln - lediglich sechs Verschraubungen gelöst werden, wenn das Fahrwerk vom Rest des Roboteraufbaus getrennt werden soll. Dies ist vorteilhaft, wenn beispielsweise Reparaturen durchgeführt werden müssen oder Modifikationen an bestimmten Roboterkomponenten vorgenommen werden. Ferner (in Abbildung 2.1 leider nicht erkennbar) besitzt jeder Roboter eine Schusseinheit mit der ein vor dem Roboter befindlicher Ball geschossen werden kann.

2.1.2 Das Fahrwerk

Bei dem verwendeten Fahrwerk handelt es sich um ein sogenanntes omnidirektionales Fahrwerk. Dieses bedeutet, dass der Roboter ohne sich zu drehen bspw. in jede beliebige Richtung fahren kann. Das Fahrwerk wurde der Projektgruppe bereits fertig überlassen, entwickelt und konstruiert wurde es im Rahmen einer Diplomarbeit von Herrn Roland Hafner[5].
Angetrieben wird das Fahrwerk von drei Motoren mit je 20 Watt Leistung, die in einem Winkel von 120 ° zueinander angeordnet sind. Die max. 6425 Umdrehungen pro Minute eines Motors werden über ein zweistufiges Planetengetriebe mit der Untersetzung von 19: 1 auf ein omnidirektionales Rad übertragen. Omnidirektional bedeutet hier, dass ein Rad nicht nur in der eigentlichen Laufrichtung des Rades, sondern auch orthogonal dazu bewegt werden kann. Realisiert wird dieses über orthogonal zur Laufrichtung des Rades angebrachte Rollen (vgl. Abbildung 2.2). Diese Rollen bestehen - wie das gesamte Rad, abgesehen von Verschraubungen - aus Polyurethan, und sind somit recht haltbar, wenngleich sie eine nicht ganz so gute Traktion auf dem Spielfelduntergrund gewährleisten. Eine denkbare Verbesserung wäre hier, die Laufrollen z. B. mit einem Gummimaterial zu beschichten.



Abbildung 2.2: Omnidirektionales Rad. Die orthogonal zur Laufrichtung angebrachten Rollen (grün) ermöglichen Bewegungen nicht nur in der eigentlichen Rad-Laufrichtung, sondern auch orthogonal zu dieser.

Ebenfalls im Fahrwerk gelagert sind die für die Stromversorgung der Motoren, des Kontrollboards (vgl. Unterabschnitte 2.3.1 und 2.3.2) und der Kamera benötigten Akkumulatoren. Bei diesen handelt es sich um drei aus je 10 Zellen bestehende Akkupacks, die je 12 V und 2400 mAh leisten. Um die Versorgungsspannung von 24 V bereitzustellen, die für die Motoren und für eine Art des Steuerboards nötig sind, sind zwei dieser Akkupacks zusammengeschaltet. Der dritte, 12 V bereitstellende Akkupack liefert den für den Betrieb der Kamera nötigen Strom. Abgesehen von der Stromversorgung über diese Akkupacks besitzt der Roboter ebenfalls noch einen Netzanschluss, über den 12 V

und 24 V eingespeist werden, um bspw. eingesetzte Akkus zu schonen.
Der generelle Aufbau des Fahrwerks besteht aus entsprechend zugeschnittenen bzw. mit den entsprechenden Winkeln versehenen Aluminiumprofilen, an denen sich mittels Einschubmöglichkeiten weitere Anbauteile befestigen lassen. So ist beispielsweise das Steuer- und Kontrollmodul, das oben auf dem Fahrwerk aufsetzt, derart befestigt.



Abbildung 2.3: Fahrwerk-Draufsicht. Erkennbar sind die im 120 °-Winkel zueinander stehenden Motoren, die die außen stehenden, omnidirektionalen Räder (grün) antreiben.

2.1.3 Steuer- und Kontrollebene

Die Steuer- und Kontrollebene dient dazu, um alle - abgesehen von der Kamera - für den Betrieb und die Steuerung des Roboters nötigen Komponenten aufzunehmen. Aufgebaut ist dieses Modul aus drei Armen, die aus je zwei Aluminiumstreben bestehen. Diese ruhen am unteren Ende in mit dem Fahrwerk verschraubten Halterungen.
Zunächst laufen diese Arme vertikal nach oben, um nach ca. 10 cm in einem 60 °-Winkel nach innen gebogen zu werden, so dass sie sich nach weiteren 30 cm treffen (vgl. Abbildung 2.1 und 2.4). Dort sind diese dann miteinander verschraubt, so dass die Konstruktion in sich sehr stabil ist.

Das pyramidenartige Zulaufen der Aluminiumstreben ergab sich aus der Verwendung eines omnidirektionalen Kamerasystems - diese Art der Konstruktion ermöglicht es der Kamera an dem Aufbau vorbei zu sehen. Somit werden von ihr auch Objekte erkannt, die sich direkt am Roboter befinden. Dieses wäre beispielsweise bei einem kastenartigen Aufbau nicht ohne weiteres gewährleistet gewesen. Ferner ist in ca. 8cm Höhe - also eben noch dort, wo die Streben vertikal laufen - eine Plattform aus Spanplatte eingesetzt, um auf dieser die diversen, nötigen Komponenten unterbringen zu können.

Konkret befinden sich in der Steuer- und Kontrollebene die folgenden Komponenten:

- das Steuerboard: zunächst war dieses das Kameleon K376SBC (vgl. Unterabschnitt 2.3.1), später wurde es ersetzt durch das Fraunhofer AIS TMC 200 - Board (vgl. Unterabschnitt 2.3.2).

- der Steuerrechner: hierbei handelt es sich um ein JVC MP-XP7210DE, ein Subnotebook mit einen Mobile Intel Pentium III mit 800 MHz, 256 MB RAM, einem SiS630ST Grafikchip und einem 8.9“Polysilizium-Flüssigkristall TFT - Bildschirm. Die Laufzeit mit einen Standardakku liegt bei ca. 3½ bis 5½ Stunden[1].

[1] vgl. auch `http://www.jvc.de/eu/mpxp/dr-datc01o.htm`



Abbildung 2.4: Komponenten der Steuer- und Kontrollebene. Im Vordergrund zu erkennen sind die drei Arme aus Aluminiumprofilen mit den entsprechenden Biegungen (an der Unterseite versehen mit den Halterungen zur Befestigung am Fahrgestell) sowie dahinter eine Trägerplatte.

- für die Schusseinheit notwendige Peripherie: der für die Druckluftversorgung der Schusseinheit nötige Drucklufttank, die Auslöseelektronik sowie das Druckventil.

2.1.4 Der Kameraaufbau

Jeder Roboter besitzt ein omnidirektionales Kamerasystem, welches einen 360°-Rundumblick ermöglicht. Dieser wird realisiert, indem die Kamera nicht direkt auf die Umgebung ausgerichtet ist, sondern auf einen Spiegel, in dem sich wegen seiner speziellen Form die gesamte Umgebung abbildet. Für genauere Informationen hinsichtlich der Funktionsweise und der Abbildung von Bildpunkten auf Weltpunkte siehe Abschnitt B.1.
Bei der in diesem System verwendeten Kamera handelt es sich um eine Firewire-Kamera der Fa. Sony, Modell DFW V500[11] mit 1/3“CCD-Chip, die bei einer Auflösung von 640*480 Bildpunkten eine maximale Bildrate von 30 fps[2] liefert. Diese Kamera ist - um den angeführten 360°-Rundumblick zu gewährleisten - senkrecht nach oben auf einen ca. 5 cm darüber befestigten hyperbolischen Glasspiegel der Firma Neovision[3] gerichtet. Dieser Spiegel bzw. diese Spiegel-Kamera-Kombination ermöglicht eine Sichtweite von ca. 5 m rund um den Roboter.

Das Problem, das sich bei der Entwicklung bzw. Konstruktion dieses Systems stellte, war die Befestigung des Spiegels über der Kamera. Die wohl stabilste Lösung wäre gewesen, ihn mittels einiger von unten nach oben über den Spiegel geführten Streben o. ä. zu befestigen, allerdings wäre dieses mit dem Nachteil verbunden, dass die Streben dann mit im Bild gewesen wären, was unter Umständen zu Problemen in der Bildverarbeitung hätte führen können. Daher wurde diese Lösung nicht weiter verfolgt, stattdessen erwies sich eine Lösung mittels einer durchsichtigen Plexiglasröhre als praktikabel. Die Kamera wurde zunächst senkrecht nach oben auf einer hölzernen Trägerplatte montiert. Anschließend wurde ebenfalls auf dieser Trägerplatte die Plexiglasröhre derart verschraubt, dass sich die Kamera mittig in dieser Röhre befindet, und der Spiegel von oben in diese Röhre eingesteckt. Um den Spiegel derart einstecken zu können, wurde er ebenfalls an einer (quadratischen) Trägerplatte montiert, deren Seitenlänge dem Außendurchmesser der Plexiglasröhre entspricht. Mittels an der Trägerplatte montierter Metallwinkel wird diese letztlich oben auf der Plexiglasröhre festgeklemmt, und mit einer Metallschelle gegen ein Verrutschen gesichert.

[2] frames per second, Bilder pro Sekunde
[3] vgl. `http://www.neovision.cz/prods/panoramic/h3g.html`



Abbildung 2.5: Montierte Spiegel-Trägerplatte. Der Spiegel ist erkennbar am unteren Ende der schwarzen Verkleidung der Plexiglasröhre, die Verkleidung dient dabei der Minimierung von Lichtreflexen in der Plexiglasröhre. Erkennbar ist außerdem die Befestigung der Trägerplatte mittels Metallwinkel.



Abbildung 2.6: Befestigung des Kameramoduls. Zu erkennen ist die Verbindung des Kameramoduls mit der Steuer- und Kontrollmodul. Ferner erkennt man ebenfalls noch einmal die Befestigung des Spiegels und die unter diesem montierte Kamera.

Bei der Konzeption und Entwicklung des Kameramoduls wurde wiederum darauf geachtet, dass sich das komplette Modul mit nicht zu großem Aufwand vom Rest des Roboter abmontieren lässt. Dieses wurde realisiert, indem die Trägerplatte lediglich mit vier Verschraubungen mit dem Roboteraufbau verbunden ist. Ferner ermöglicht diese Art der Verschraubung eine Justierung des kompletten Kameramoduls, optimalerweise muss die Kamera-Spiegel-Kombination natürlich exakt senkrecht angeordnet sein.

Insgesamt erwies sich dieser Kamera-Spiegel-Aufbau nicht als optimal, beispielsweise schwingt oder wackelt dieser Aufbau recht stark wenn der Roboter fährt, aber für die an ihn gestellten Anforderungen erwies er sich als ausreichend.

2.2 Steuerrechner

Auf dem Steuerrechner läuft die eigentliche Applikation, welche neben der Ansteuerung des Controller-Boards z.B. die Objekterkennung oder die Pfadplanung durchführt. Als Steuerrechner kann jeder Standard-PC oder mobile Rechner mit einer seriellen Schnittstelle und einem Firewire-Port eingesetzt werden. Für das Ziel eines autonomen Roboters verbietet sich die Lösung durch Standard-PCs aus mehreren Gründen, so dass verstärkt nach Notebooks gesucht wurde, welche den Anforderungen

genügten. Kritikpunkte waren u.a. diese:

- Stromversorgung: Auf dem Roboter selbst befinden sich zwei getrennte Stromkreise, einer mit 12 V DC und einer mit 24 V DC. Moderne Rechner sind mit sog. ATX-Boards ausgestattet. Setzt man solche ein, dann erfordert dies die Bereitstellung weitaus vielfältigerer Spannungen wie +3.3 V, +5 V, -5 V, +12 V sowie -12 V und einen erhöhten Hardwareaufwand. Gleichzeitig wird mit dem ATX-Board ein neuer Verbraucher an die Akkumulatoren gehängt, dessen Leistungsaufnahme im Bereich zwischen 150 und 250 Watt liegt. Verglichen mit den maximal 20 Watt pro Motor, lässt sich die Unverhältnismäßigkeit der Mittel schnell erkennen.

- Maße: Die Maße eines ATX-Boards belaufen sich auf ca. 30.5x24.4 cm (LxB). Als Referenz dient das K7S5A Mainboard der Firma Elitegroup, auf aktuelleren Boards sind bereits Grafikkarten integriert, womit die Gesamthöhe einer ATX-Lösung mit nicht mehr als 5 cm abgeschätzt wird. Der Einsatz eines off-the-shelf Notebooks würde zu den gleichen Ausmaßen führen, jedoch sofort das Problem der Stromversorgung lösen.

- Peripherie: wird eine Lösung durch Standard-PCs verfolgt so ergibt sich zwangsläufig die Frage, welche weiteren peripheren Geräte zur Arbeit am Roboter benötigt werden: neben einer Maus und der Tastatur zur Eingabe fehlt ein Display zur Datenausgabe. Das Display an sich bedarf einer Stromversorgung, soll diese über die Akkus möglich sein? Wenn nein, dann schränkt die Stromleitung des Bildschirms den Aktions- bzw. Arbeitsradius ein. Wenn doch, dann kämen nur spezielle 12 V bzw. 24 V TFT-Bildschirme in Frage. Offenbar findet sich auch unter diesem Aspekt in der Verwendung eines Notebooks eine Lösung, da darin bereits alle Komponenten integriert sind.

Nicht nur Standard-PCs standen auf dem Prüfstand, weitere Überlegungen gingen in die Richtung von ITX-Boards und PC104-Stecksystemen.

Heutige Notebooks bieten durch die Verbreitung des USB-Standards nicht immer eine serielle



Abbildung 2.7: Der Steuerrechner JVC MP-XP7210

Quelle: `http://www.jvc.de`

Schnittstelle an und wenn doch sind die Notebooks durch ihre Ausmaße für den Einsatz auf unserer mobilen Plattform nicht geeignet.
Das Sub-Notebook MP-XP7210 der Firma JVC wurde aufgrund seiner geringen Maße von 225 x 29.5 x 177 mm (BxHxT) und des Gewichtes von nur ca. 1050 g ausgewählt. Kaum größer als ein DINA-5 Blatt ist es geradezu prädestiniert für unsere Zwecke. Wie aber für Notebooks dieser Größenordnung zu erwarten, fehlt eine serielle Schnittstelle (RS232), so dass eine Ersatzlösung durch einen PCMCIA-2-Serial-Adapter der Firma Elan gewählt wurde.

CPU	Pentium III-M, 800 MHz
Hauptspeicher	256 MByte
Festplatte	30 GByte
Schnittstellen	USB, IEEE1394, PCMCIA

Tabelle 2.1: Technische Daten des JVC MP-XP7210

2.3 Motorcontroller

Der Motorcontroller bildet das Herzstück des Roboters, er ist verantwortlich für die Ansteuerung der einzelnen Räder. Man könnte einwenden, dass der Steuerrechner die Fahrbefehle ausgibt, jedoch sind etliche der Motorcontroller programmierbar, so dass für einfache Aufgaben wie das Abfahren einer Figur ein Steuerrechner komplett entfallen kann, da die Fahrbefehle auf dem Motorcontroller selbst erzeugt und verarbeitet werden können.

Die Komponenten auf den diversen Controllern variieren je nach Preisklasse in Qualität und Anzahl, z.B. zusätzliche Möglichkeiten zur Verarbeitung von E/A oder die Regelgüte bei einer Drehzahlregelung. Einige Komponenten sind aber auf fast allen Motorcontrollern wiederzufinden:

- Kommunikationsschnittstelle: die auf dem Motorcontroller vorhandene Rechenkraft ist sehr stark begrenzt, aufwendigere Aufgaben wie Odometrieberechnungen und Pfadplanung lassen sich damit nur unzureichend lösen. Eine externe Quelle (z.B. ein Laptop) übernimmt diese Aufgabe und schickt die daraus resultierenden Ergebnisse bzw. Fahrbefehle an den Motorcontroller. Für die Kommunikation zwischen den beiden Geräten gibt es mehrere Möglichkeiten, im einfachsten Fall ist es eine drahtgebundene Kommunikation über eine serielle Schnittstelle oder einen CAN-Bus[4]. Im oberen Preissegment finden sich aber ebenso Controller, die über eine drahtlose Verbindung wie Bluetooth mit dem Host kommunizieren können.

- Mikrocontroller: er ist das Kernstück des Motorcontrollers, durch seine Auswahl wird u.a. die Leistungsfähigkeit bestimmt. Verbunden mit der Kommunikationsschnittstelle nimmt er Befehle entgegen und wandelt diese z.B. in PWM-Signale zur Motorsteuerung um oder liest einen A/D-Eingang aus.

- Leistungselektronik: die eingesetzten Motoren können nicht direkt vom Mikrocontroller angesteuert werden, da dessen Ausgangsleistung einfach zu schwach ist. Daher wird noch zusätzliche Elektronik zwischen Motor und Mikrocontroller eingefügt.

Während des Verlaufs der Projektgruppe kamen zwei verschiedene Controller zum Einsatz: das Kameleon 376SBC stand seit Beginn der Projektgruppe zur Verfügung und war bereits während einer Diplomarbeit[5] verwendet worden. Nach den German Open 2003 stellte das Fraunhofer-Institut für autonome intelligente Systeme uns auf Nachfrage einen Prototypen ihres selbstentwickelten Motorcontrollers TMC200 zur Verfügung.

2.3.1 Kameleon 376SBC

Das Kameleon 376SBC (siehe Abbildung 2.8) wurde von der Firma K-Team hergestellt, mittlerweile ist jedoch die Produktreihe eingestellt worden, ein Nachfolger steht mit dem *Korebot* kurz vor der Markteinführung. Den Kern des Boards bildet ein MC68376 Mikrocontroller mit einer Taktfrequenz von 20 MHz.

[4] Controller Area Network, vgl. `www.can.bosch.com`



Abbildung 2.8: Das Motorcontroller-Board Kameleon 376SBC

Quelle: `www.k-team.com`

Mikrocontroller	MC68376
RAM	1 MByte
Flash-Speicher	1 MByte
Spannungsversorgung	8.5 - 20 V
Leistungsaufnahme	< 1 Watt
Kommunikation	RS-232, K-Net, MMA

Tabelle 2.2: Auswahl technischer Daten zum Kameleon 376SBC

Durch die Bereitstellung verschiedener Bussysteme wie SXB[5], CXB[6] oder IXB[7] bietet das Board vielfältige Erweiterungsmöglichkeiten, insbesondere ist die Zusammenschaltung mehrerer Kameleon-Boards möglich. Der firmenspezifische Bus K-Net ermöglicht den Anschluss vorgefertigter Sensormodule.

Auf dem Board selbst läuft ein echtzeitnahes Betriebssystem, das den Zugriff auf alle Ressourcen über BIOS-Aufrufe erlaubt. Ein kostenlos bereitgestellter Crosscompiler ermöglicht das Erzeugen eigener Anwendungen, die anschließend in einem nichtflüchtigen Speicher auf dem Board gespeichert werden können. Über einen DIP-Schalter lassen sich verschiedene Boot-Modi einstellen, einer von ihnen ermöglicht die Ausführung eines vorher im Flash gespeicherten Programms und ein anderer die Nutzung eines fest einprogrammierten Kommunikationsprotokolls über die serielle Schnittstelle. Der Steuerrechner kann im zweiten Boot-Modus somit direkt Befehle an das Board senden und so Motorgeschwindigkeiten setzen oder Sensorinformationen abfragen.

Eine Erweiterungsplatine, das REB[8] (siehe Abbildung 2.9), die auf das Kameleon aufgesteckt wird, enthält die notwendige Leistungselektronik zur Ansteuerung von bis zu vier Motoren mit jeweils maximal 20 Watt. Zur Erfassung der Motordrehzahl stellt die Platine Anschlüsse für Inkrementalencoder bereit, der onboard-PID-Controller wertet diese Informationen z.B. zur Drehzahlregelung aus. Zusätzlich bietet es eine Vielzahl an Schnittstellen für Sensoren und zur Ansteuerung weiterer Aktoren wie Servos, so können über einen Teil der analogen E/A-Pins die aktuelle Versorgungsspannung und die Motorströme ausgelesen werden.

[5] serial extension bus
[6] cpu extension bus
[7] interface extension bus
[8] Robotic Extension Board

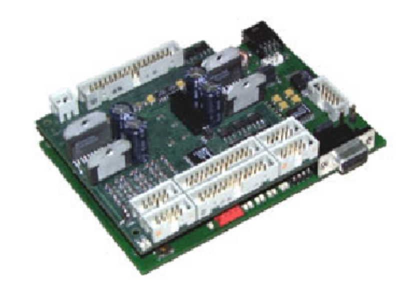


Abbildung 2.9: Das Robotic Extension Board für das Kameleon 376SBC

Quelle: `http://www.k-team.com`

Motion	4 DC Motoren & Encoder 4 Servo-Ausgänge
I/O	11 A/D-Wandler, 10 Bit Auflösung 2 D/A-Wandler, 12 Bit Auflösung 16 digitale Kanäle
Konfigurierbare I/O	16 digitale I/O für Sensorik
Schnittstellen	I2C-Bus

Tabelle 2.3: Auswahl technischer Daten des Robotic Expansion Boards

2.3.2 TMC200

Der Triple-Motor-Controller, kurz TMC200, (siehe Abbildung 2.10) ist eine Entwicklung des Fraunhofer Instituts für autonome intelligente Systeme und zur Zeit noch in einem Teststadium was Hard- und Software angeht. Er ist für bis zu drei Gleichstrommotoren mit einer maximalen Leistung von 200 Watt konzipiert. Den Boardkern bildet ein 16-Bit Mikrocontroller von Infineon. Durch den eingebauten PID-Regler ist eine Drehzahlregelung mit hohen Anforderungen an Gleichlauf und geringen Drehmomentschwankungen möglich. Eine Besonderheit ist der eingebaute thermische Motorschutz durch eine I^2T-Strombegrenzung: Die Rotorwicklungstemperatur ist von außen nicht messbar und wird über ein thermisches Modell berechnet; überschreitet die Temperatur einen kritischen Wert droht ein Wicklungsbrand, der den Motor zerstört. Innerhalb des Motorcontrollers wird dazu das thermische Modell als nebenläufiger Prozess ausgeführt, der beim Überschreiten der kritischen Wicklungstemperatur eine Strombegrenzung aktiviert. Die bisherige Obergrenze für den zulässigen Strom wird vom Anlaufstrom abgesenkt auf den motorspezifischen maximalen Dauerstrom, gleichzeitig beginnt die Abkühlung des Motor auf seine Nenntemperatur bzw. je nach Belastungsgrad auf einen niedrigeren Wert. Erreicht die Modelltemperatur eine bestimmte untere Grenze, so wird die Strombegrenzung deaktiviert, so dass dem Motor erneut der Anlaufstrom als Obergrenze zur Verfügung steht.

Ein frei nutzbarer 10 Bit-E/A-Port dient zum Anschluss eigener Sensorik, wobei Sensorik und Ports am besten optisch entkoppelt sind. Die Kommunikation mit dem Steuerrechner erfolgt im Moment ausschließlich über die serielle Schnittstelle durch den Austausch von ASCII-Nachrichten, die Unterstützung des CAN-Bus ist board-seitig noch nicht implementiert.



Abbildung 2.10: Der Motorcontroller TMC200

Quelle: `http://www.ais.fraunhofer.de`

Mikrocontroller	Infineon C164CI
Spannungsversorgung	18 - 30 V
Leistungsaufnahme	< 1.5 Watt
Kommunikation	RS-232, CAN
E/A	3 Motoren & Encoder 10 A/D-Wandler, 8 Bit Auflösung

Tabelle 2.4: Auswahl technischer Daten zum TMC200

2.4 Schusseinheit

Zu einem kompletten Fußballspieler gehört neben seiner Bewegungsfreiheit und der Fähigkeit Entscheidungen zu treffen eine Möglichkeit zur Ballbehandlung, z.B. einem Dribbling oder einem Schuss. Der Mensch setzt hierfür seine Beine bestehend aus Muskeln, Sehnen und Knochen ein. Für Roboter muss eine andere Lösung angestrebt werden, da die Technik auf diesem Gebiet noch nicht so weit fortgeschritten ist: so gibt es z.B. Nanomuskeln, deren Kraft für unsere Zwecke aber bei weitem nicht ausreichend ist.
Der verfolgte Ansatz orientiert sich an der Natur des Menschen und bildet mit einer Konstruktion aus Aluminiumteilen ein Stück des Fußes nach (siehe Abbildung 2.11). Ein fest mit dem Roboter verschraubtes Aluminiumelement bildet das “Standbein“. Der bewegliche Teil der Schusseinheit ist über ein Scharnier mit dem oberen Ende des Aluminiumelements verbunden, vergleichbar mit dem Hüftgelenk. Das Fehlen eines Muskels kompensiert der Einsatz eines Pneumatikzylinders, der wie in Abbildung 2.11 angebracht ist. Die Schusseinheit des Roboters wird pneumatisch betrieben, aus Abbildung 2.12 kann man die einzelnen Komponenten entnehmen. Der eingesetzte Druckluftbehälter kann mit bis zu 10 bar betankt werden. Um das Rückfließen der Luft zu verhindern setzen wir ein Rückschlagventil ein, welches durch einen Aufsatz mit einem Kompressor bzw. einer Handluftpumpe (Autoventil) verbunden wird. Die zweite Öffnung des Behälters dient als Auslass und ist über einen Druckluftschlauch (∅ 8 mm) mit einem Drosselrückschlagventil verbunden. Dieses Ventil kann für eine grobe Regulierung der Durchflußmenge manuell geöffnet bzw. geschlossen werden.

Von dort aus wird die Druckluft auf ein 3/2-Wege-Magnetventil weitergeleitet, d.h. es besitzt drei Öffnungen und zwei definierte Schaltstellungen. Die Öffnungen sind gekennzeichnet durch die Buchstaben

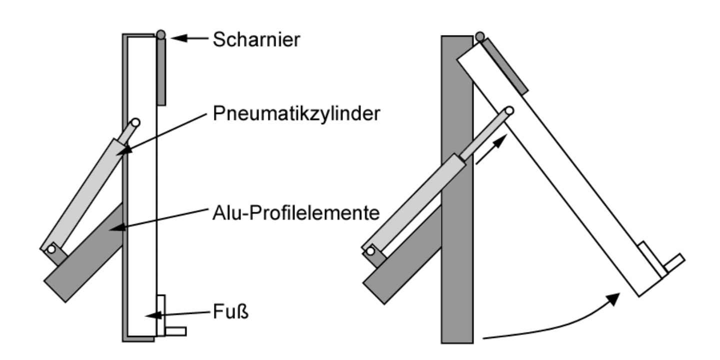


Abbildung 2.11: Mechanischer Aufbau der Schusseinheit. Links im eingefahrenen und rechts im ausgefahrenen Zustand.

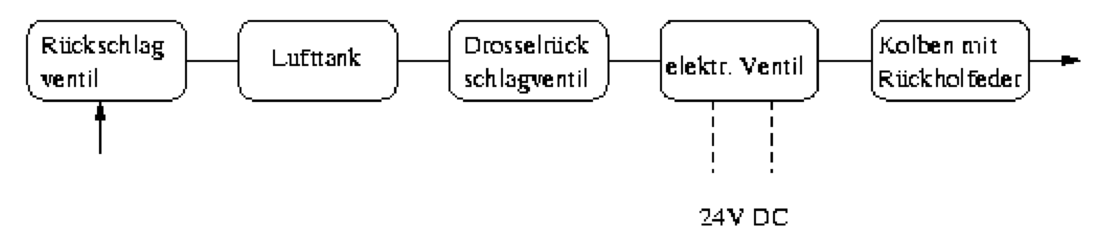


Abbildung 2.12: Schematische Darstellung des Zusammenspiels der Pneumatik-Elemente

- P: Druckluftanschluss
- R: Belüftung
- A: Anschluss für einen Zylinder

und entsprechend „beschalten“.

In der einen definierten Stellung des Ventils sind zur Entlüftung des Zylinders A und R verbunden, dies ist die Standardstellung: der Kolben des Zylinders ist eingezogen. Befindet sich das Ventil in der anderen Stellung, so sind die Ventilöffnungen P und A verbunden. Der Zylinder füllt sich mit Druckluft, welche den Kolben nach vorn treibt.

Wie oben angedeutet handelt es sich hierbei um ein Magnetventil, die von außen einwirkende Größe ist eine elektrische Spannung (24 V DC). Durch Anlegen bzw. Abschalten der Spannung wandert der Magnet innerhalb des Ventils zwischen den definierten Positionen.

Der eingesetzte Rundzylinder besitzt einen Kolben mit Rückholfeder. Das Einfahren des Kolbens bei der Zylinderentlüftung erfolgt somit rein durch Federkraft. Im Gegensatz hierzu gibt es zweifachwirkende Zylinder, bei denen für Ein- und Ausfahren jeweils Druckluft verwendet wird. Unter Verwendung dieses Zylindertyps hätte sich die Anzahl der möglichen Schüsse und die jeweilige Schusskraft stark reduziert.

Die aktuelle Lösung ist bei weitem nicht optimal: bei einem Schuss bremst das Spannen der Rückholfeder den beschleunigenden Kolben ab. Die maximale Schussgeschwindigkeit wird damit nie erreicht.

Zur Auslösung der Schusseinheit sind mehrere Möglichkeiten in Betracht gezogen und getestet worden, die nachfolgend in ihrer zeitlichen Reihenfolge beschrieben werden.

- Mikroschalter: Als „Drucksensoren“ befestigten wir zwei Mikroschalter (mit Federbügel) an die Innenseiten der Ballführung. Sie wurden so ausgerichtet, dass die Schusseinheit bei einem Ballabstand von ca. 5 cm zum Roboter auslöste. Das Auslesen der Sensoren erfolgte über den Motorcontroller K376SBC in einer Zykluszeit von 10 ms. Ein irrtümliches Auslösen, z.B. wenn ein gegnerischer Roboter gegen die Mikroschalter fährt, verhindert eine zusätzliche Sperre. Per Software setzbar basierte sie auf Informationen bzgl. der Ballposition aus dem Weltbild.

- Ultraschall: Durch Testspiele und die German Open 2003 zeigten sich schnell Schwächen im Einsatz der Mikroschalter. Bereits nach einigen Kollisionen lösten die Mikroschalter durch verbogene Federbügel nicht mehr zuverlässig aus. Die berührungslose Messung mittels eines oder zweier Ultraschallsensoren, die an der Front des Roboters schräg hinab blicken, sollte Besserung bringen. In Tests mit einem separaten Mikrocontroller-Board (C-Controll II[9]) ergaben sich Messungenauigkeiten von maximal einem Zentimeter. Dieses Ergebnis ließ sich nicht auf den bis dahin verwendeten Motorcontroller K376SBC übertragen, er verfügt über die benötigte I^2C-Schnittstelle, deren Implementierung und Funktionalität war aber nicht ausgereift.

- Kamerabild: Das Kamerabild an sich bietet sehr viele Informationen und zusammen mit der Spiegelgleichung sind Entfernungen im Nahbereich gut abschätzbar. Wie schon bei den Mikroschaltern betrachtet man die Entfernung des Balls vom Roboter, ab einem bestimmen Pegel löst der Schuss aus. Jedoch fällt hier eine Latenzzeit von ca. 30-60 Millisekunden an, so dass der Ball bereits weiter gerollt sein kann und der Schuss ins Leere geht. Zur Zeit wird diese Methode verwendet.

- Fotowiderstände: Zwei Fotowiderstände wurden leicht schräg nach oben schauend an der Roboterfront montiert. Befindet sich der Ball nah genug an der Front, verdeckt sein Schatten das auf die Widerstände fallende Licht. Die daraus resultierende Widerstandsänderung wird über einen PIC[10] ausgewertet und die Schusseinheit ausgelöst. Wie bei den Mikroschaltern gibt es eine Software-Sperre, die das irrtümliche Auslösen vermeidet.
  Diese Variante wurde als letzte getestet, verwertbare Ergebnisse stehen aus.

[9] `http://www.c2net.de`
[10] programmable IC

Kapitel 3

Software

3.1 Überblick

Dieses Kapitel beschreibt die Software, die die Projektgruppe im Laufe der Projektarbeit entwickelt und eingesetzt hat. Die Software, die zum Betrieb der Roboter benötigt und entwickelt wurde, gliedert sich in zwei wesentliche Stränge. Zum einen gibt es auf jedem Roboter das sogenannte Steuerprogramm - dieses wird in den folgenden Abschnitten 3.2 - 3.6 beschrieben. Zum anderen existiert ein Kommunikationsprogramm, das auf einem zentralen Kommunikationsrechner außerhalb des Spielfelds läuft. Die dazugehörige Beschreibung und Bedienungsanleitung findet sich im Anhang A.3. Bevor wir nun wie angekündigt in späteren Abschnitten dieses Kapitels in die detaillierte Beschreibung der Software einsteigen, geben wir an dieser Stelle einen kurzen Überblick über die einzelnen Module, aus denen sich die Steuerungssoftware der Roboter aufbaut. Wir betrachten deren Funktionsweise sowie den Kontrollfluss in der Applikation, siehe Bild 3.1, und zeigen auf, welche Komponenten von welchen Ausgaben anderer Komponenten abhängig sind. Anschließend betrachten wir die Programmablaufsteuerung.

3.1.1 Aufbau und Funktionsweise der Module

Die einzelnen Module der Software arbeiten wie in Abbildung 3.1 dargestellt zusammen. Die Komponenten werden jeweils sequentiell d.h. nacheinander abgearbeitet. Dabei bauen zeitlich später arbeitende Module auf den Ausgaben der zeitlich vorher liegenden Module auf. Abbildung 3.2 beschreibt, wie Daten zwischen den Modulen bei der sequentiellen Abarbeitung durchgereicht werden. An vorderster Stelle wird immer die Bildverarbeitung durchgeführt, da die Kamera-Sensorinformationen die Grundlage für alle weiterführenden Berechnungen bilden.
Die Funktionalität der einzelnen Module mit den zugehörigen Eingabedaten und Ausgabedaten ist dabei wie folgt:

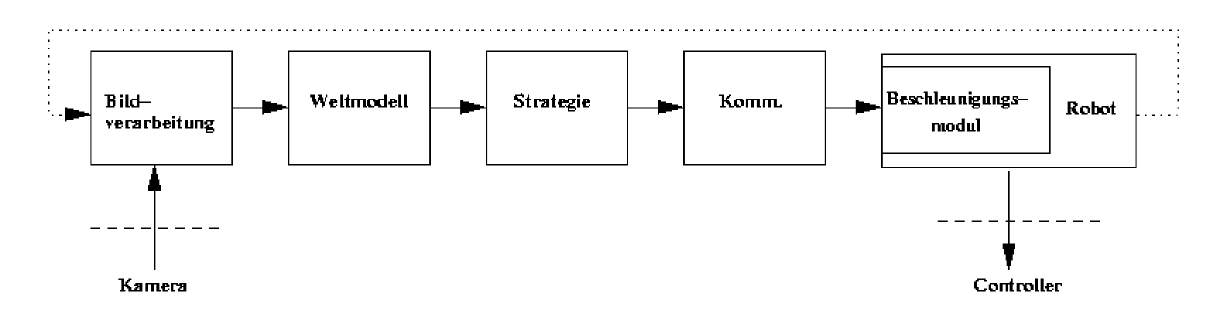


Abbildung 3.1: Sequentieller Kontrollfluss in der Softwarearchitektur

- **Bildverarbeitung** (siehe Abschnitt 3.2)
  Eingabe: Bild der Kamera

  Die Bildverarbeitung erhält als Eingabe ein Bild der Kamera übergeben und verarbeitet dieses, indem alle interessanten Regionen im Bild herausgefiltert werden. Diese Regionen entsprechen im physischen Umfeld Objekten wie Tore, Ball, Gegner usw. Gekennzeichnet sind sie typischerweise durch eine fest vorgegebene Farbgebung[1]. Zur Ausgabe aus dem Bildverarbeitungsmodul gehören neben den Winkeln zu den erkannten Objekten auch die Entfernungen, die sowohl in Pixeln im Bild als auch in Längeneinheiten (cm, mm) gegeben sein können. Der Roboter ist dabei Bezugspunkt aller Berechnungen hinsichtlich Entfernungsangaben und Winkel.

  Ausgabe: interessante Regionen im Bild mit robozentrischen Winkeln und Entfernungen

- **Weltmodell** (siehe Abschnitt 3.3)
  Eingabe: Regionen der Bildverarbeitung, Sensorinformationen über die physische Bewegung des Roboters

  Das Weltmodell ist verantwortlich für die Erstellung eines Modells, einer Karte der Umwelt. Aus der Kenntnis der Position und Entfernung zu markanten und festen Punkten im Spielfeld wie Toren oder Eckfahnen (Eingabedaten) wird die Position des Roboters auf dem Spielfeld eindeutig bestimmt. Weiterhin sind im Modell neben der eigenen Position auch die Positionen von Gegnern, dem Ball usw. enthalten. Dabei werden die interessanten Regionen von relativen auf absolute Positionen und Winkel umgerechnet.

  Ausgabe: Regionen versehen mit absoluten Winkeln und Positionen

- **Strategiemodul** (siehe Abschnitt 3.4)
  Eingabe: absolutes Weltmodell, robozentrische Regionen der Bildverarbeitung, Anweisungen des zentralen Kommunikationsrechners zum Rollenwechsel

  Im Strategiemodul werden passend zur aktuellen Situation die Aktionen berechnet, die der Roboter durchzuführen hat, um ein bestimmtes Ziel zu erreichen. Dabei wird die Rolle wie Verteidiger, Angreifer oder Torwart berücksichtigt. Dazu gehörige Ziele sind beispielsweise Verteidigung des Torraums, Verteidigung der eigenen Hälfte, oder Angriff auf das gegnerische Tor. Eine wählbare Aktion könnte sein, dass ein Verteidiger in seiner Hälfte patrouilliert, um angreifende Gegner abzuwehren. Dabei ist in Abhängigkeit von den Eingabedaten zu beachten: Wenn die Güte der Selbstlokalisation unter einen Schwellwert fällt, kann das Weltmodell nicht als Eingabe zur Berechnung einer Aktion akzeptiert werden. Statt dessen werden die Regionen der Bildverarbeitung benutzt. Aus den vorliegenden Informationen zur Umgebung - Richtung und Entfernung zu Hindernissen, Toren und Ball - und den lokalen Randbedingungen des Roboters wie Maximalgeschwindigkeit wird ein Fahrbefehl berechnet. Falls der zentrale Kommunikationsrechner einen Rollenwechsel verlangt, so fließt dies ebenfalls bei der Berechnung der Ansteuerung mit ein.

  Ausgabe: Ansteuerungsbefehl

- **Kommunikationsmodul** (siehe Abschnitt 3.5)
  Eingabe: Anweisung zum Rollenwechsel

  Das Kommunikationsmodul kommuniziert mit einem zentralen Rechner, der zusätzlich zu den lokal abgeschlossenen Einheiten der Roboter außerhalb des Spielfelds zum Einsatz kommt. Diese Zentrale bündelt und verarbeitet alle Status- und Positionsinformationen, die sie kontinuierlich von allen Robotern erhält. Umgekehrt nimmt das Kommunikationsmodul Anweisungen von dem zentralen Kommunikationsrechner entgegen. Solch eine Anweisung betrifft die Aufforderung zum Rollenwechsel an die Roboter verbunden mit einem Verhaltenswechsel. Das heißt, falls einer der Angreifer ausfällt, so wird entschieden, welcher Roboter nun jeweils die offene Rolle einnehmen soll. Beispielsweise muss Verteidiger 1 an dessen Stelle treten, und Verteidiger

2 übernimmt nun zusätzlich das Gebiet von Verteidiger 1.
Die Komponente fungiert also als Schnittstelle zwischen Roboter und zentralem Kommunikationsrechner. Die Informationen zur selbstlokalisierten Position sowie dem Status[1] des Roboters werden an den Kommunikationsrechner weitergereicht. Die Aufforderung zum Rollenwechsel wird an die Strategie weitergeleitet.

Ausgabe: Anweisung zum Rollenwechsel, Selbstlokalisationsposition/Statusinformation

- **Beschleunigungsmodul** (siehe Abschnitt 3.6.3)
  Eingabe: Ansteuerungsbefehl der Strategie

  Im diesem Untermodul zum Robotermodul (siehe nächster Punkt) wird der Ansteuerungsbefehl aus der Strategie vor der eigentlichen Übersetzung in der Roboterkomponente vorverarbeitet. Physikalisches Wissens zum Roboter steht hier zur Verfügung, das bei der Strategieberechnung nicht berücksichtigt werden muss. Die Ansteuerungsbefehle werde an reale Begebenheiten wie die Gefahr des Umkippens angepasst. Dies kann zum Beispiel geschehen, wenn der Roboter an zwei aufeinanderfolgenden Zeitpunkten widersprüchliche Ansteuerungen erhält. Beispielsweise: fahre zum Zeitpunkt t mit Maximalgeschwindigkeit in Richtung $0\,^\circ$ Grad, zum Zeitpunkt $t+1$ mit Maximalgeschwindigkeit in die entgegengesetzte Richtung. Im konkret beschriebenen Fall wird der Roboter in eine Kurvenfahrt übergehen, um den harten Wechsel zwischen den beiden Ansteuerungsbefehlen auszugleichen.

  Ausgabe: adaptierter Ansteuerungsbefehl

- **Robotermodul** (siehe Abschnitt 3.6)
  Eingabe: adaptierter Ansteuerungsbefehl der Strategie

  An allerletzter Stelle im Kontrollfluss der Software verarbeitet das Robotermodul den Ansteuerungsbefehl an die Hardwareansteuerung des Roboters. Dabei leistet es die Übersetzung eines auf hohem Niveau formulierten Ansteuerungsbefehls an die spezifische Hardwareplattform des Roboters, da beispielsweise mehrere Motorcontroller am selben Roboter betrieben werden können .
  Das Modul, das an der Schnittstelle zwischen Applikation und Betriebssystem bzw. Roboterhardware steht, schickt den Ansteuerungsbefehl an die serielle Schnittstelle des Steuerrechners, mit dem der Motorcontroller des Roboters verbunden ist. Weiterhin nimmt es Sensorinformationen des Roboters entgegen und reicht diese an das Weltmodell weiter.

  Ausgabe: Ansteuerungsbefehl in Form von Befehlen an die einzelnen Räder des Roboters, Sensorinformationen (Odometriedaten)

3.1.2 Programmablaufsteuerung

Das benutzte Betriebssystem auf dem Steuerrechner ist ein Realtime-Linux. Ein Linux ohne Echtzeitunterstützung garantiert bei der Abarbeitung von beispielsweise der Kommunikation an der seriellen Schnittstelle nur eine Genauigkeit von ca. 10 ms[2]. Ein Fahrbefehl, der über die serielle Schnittstelle geschickt wird, würde gesichert erst nach 10 ms ankommen. Diese Zeitspanne ist für die echtzeitnahe Kontrolle eines Roboters jedoch zu groß. Demgegenüber biete das echtzeitnahe RTLinux[3] eine Genauigkeit von ca. 1 ms. Mehr dazu auf Seite 56 im Abschnitt über das Roboterkommunikationsmodul.

In unserer Architektur, mit den eingesetzten Algorithmen hat es sich gezeigt, dass die Programmschleife (siehe Algorithmus 1) mit einer Geschwindigkeit von $\Delta_t = 60\,\text{ms}$ laufen kann, was einer

[1] funktionsfähig/nicht funktionsfähig
[2] Dies entspricht der Standardeinstellung eines Linux. Durch verschiedene Einstellungen kann die Genauigkeit erhöht werden, jedoch besteht hier die große Gefahr von unkontrollierbaren Nebenwirkungen.
[3] http://www.rtlinux.org

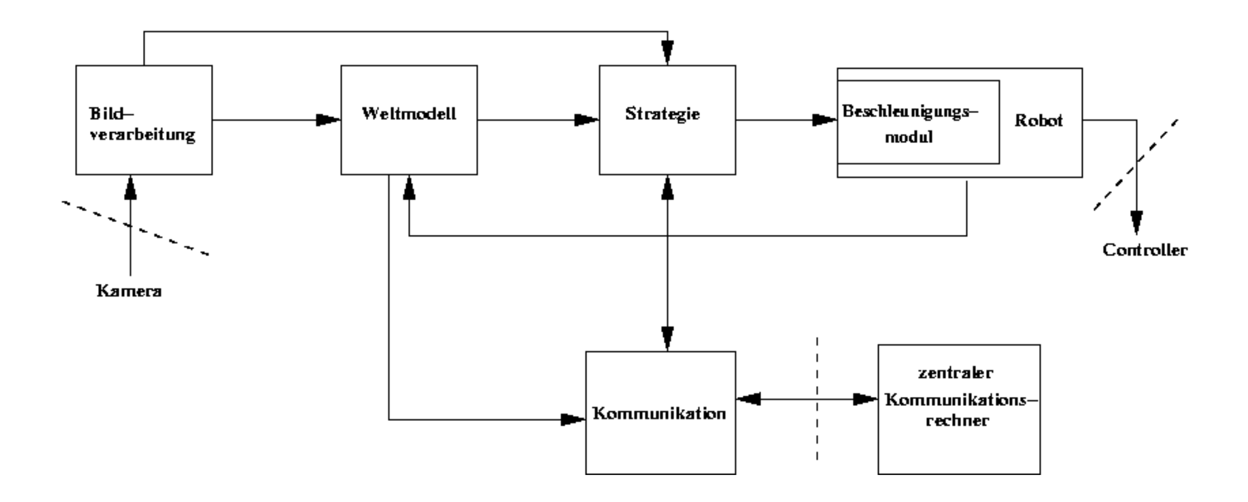


Abbildung 3.2: Datenfluss in der Softwarearchitektur

Frequenz von ungefähr 17 Hz entspricht. Für den Kontrollfluss im Programmablauf wurde die genaue Zeitmessung des RTLinux eingesetzt. Damit wurde erreicht, dass zwischen zwei vollständigen Durchläufen der Schleife in der Regel $\Delta_t = 60\,\text{ms}$ vergehen. Sofern der Durchlauf schon vor Ablauf der eingestellten Zykluszeit fertig ist, wird auf den Ablauf eines gesetzten Timers gewartet. Diese Einschränkung an den Programmfluss wird bisher hauptsächlich für die Berechnung der Ballgeschwindigkeit beim Torwart (siehe Unterabschnitt 3.4.3) eingesetzt. Andere Module benutzen diese Informationen implizit. Beispielsweise müssten die Einstellungen des Beschleunigungsmodul adaptiert werden, wenn sich die Zykluszeit ändert. Der Motorcontroller des Roboters kann in einer Zeit von $\Delta_t = 60\,\text{ms}$ gefahrlos andere Geschwindigkeitsdifferenzen ausregeln als in einer Zeitspanne von $\Delta_t = 100\,\text{ms}$. Entsprechend wird der Roboter bei unterschiedlichen Fahrmanövern ungewünschtes Verhalten zeigen, das durch das Beschleunigungsmodul abgefangen werden muss.

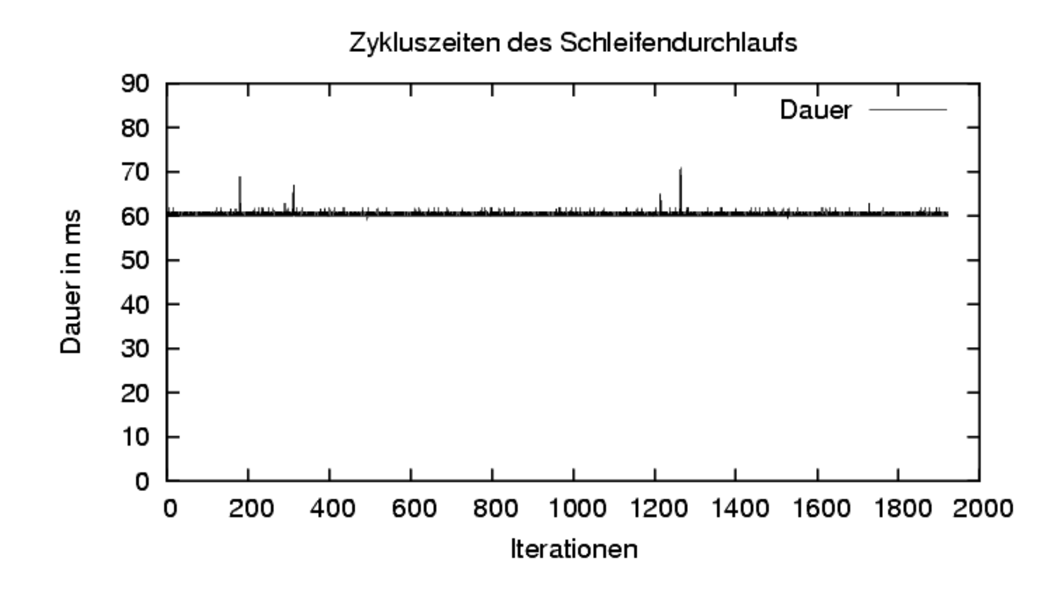


Abbildung 3.3: Zyklusdauer der Hauptschleife von Tribots

Ein Durchlauf von dem Holen und Verarbeiten des Bildes bis zum Ansteuern des Roboters dauert also in der Regel nicht mehr als $\Delta_t = 60\,\text{ms}$. In Bild 3.3 sind kleinere Ausschläge gegenüber der 60 ms-Linie zu sehen. Es ist bisher unbekannt, woher diese Ungenauigkeiten stammen. Wir ver-

muten jedoch, dass hier das Betriebssystem wiederkehrend anderen Prozessen zuviel Rechenzeit zur Verfügung stellt, so dass unser Programm nicht früh genug die Kontrolle zurückerhält, um nach Ablauf des des Schleifentimers einen neuen Zyklus einzuleiten.

Modul	Dauer (in ms)
Bildverarbeitung und Weltmodell[4]	27
Strategie	2
Kommunikation	1
Robotermodul[5] (inkl. Beschleunigungsmodul)	17 (stehend) 20 (fahrend)
GUI	3
inaktiv	$\Delta_t = 60\,\mathrm{ms} - restl.Modulzeiten$ 7

Tabelle 3.1: Durchlaufzeiten der einzelnen Softwaremodule

Der große Vorteil der sequentiellen Programmablaufsteuerung gegenüber einer parallelen Abarbeitung der Module liegt darin, dass der Aufwand der Synchronisierung von Threads und der zeitlichen Einordnung von Daten (z.B. Objektdaten aus einem Bildverarbeitungsthreads) schlicht entfällt. Die Module arbeiten immer genau dann, wenn neue Informationen für sie vorliegen, wodurch alle Rechenzeit dem gerade rechnenden Modul zur Verfügung steht.
Tabelle 3.1 zeigt die Zeit an, die die Module im Mittel für die Berechnungen benötigen. Größere Abweichungen können unter anderem dann vorkommen, wenn in der Bildverarbeitung bei der Kalibrierung der Farbeinstellung ein Bild angezeigt werden muss. In diesem Falle nimmt das Zeichnen der grafischen Oberfläche übermäßig viel Zeit in Anspruch. Darüber hinaus laufen die Algorithmen stabil.

1: **while** Programm läuft **do**
2: block($\Delta_t = 60\,\mathrm{ms}$); {Start des RT-Linux-Timer}
3: Weltmodell→compute_actual_wmstate(); {Aktuelles Bild der Kamera wird geholt und bearbeitet. Darauf werden Berechnungen im Weltmodell durchgeführt.}
4: Strategiemodul→compute_decision(); {Berechnungen der Strategie: es wird eine Aktion und ein Ansteuerungsbefehl bestimmt.}
5: Kommunikationmodul→processMessages(); {Kommunikation mit dem zentralen Kommunikationsrechner}
6: Beschleunigungsmodul→driveXYPVelocity(); {Anpassung der Ansteuerung durch Beschleunigungsmodell, Abschicken des Befehls an den Roboter}
7: GUI→processEvents(50); {GUI erhält 50ms Zeit für alle grafischen Updates.}
8: wait_for_deblock(); {Falls Schleifen-Zeit nicht überschritten wird hier auf den Ablauf des Timer gewartet.}
9: **end while**

Algorithmus 1: Hauptschleife der Tribots-Anwendung in Pseudocode

[5]Wie Algorithmus 1 zeigt, ruft das Weltmodell das Holen und Verarbeiten eines Bildes in der Bildverarbeitung auf. Die Messung erfolgt daher im Verbund.

[6]Gemessen mit dem TMC. Die unterschiedlichen Durchlaufzeiten ergeben sich, weil durch die Bewegung mehr Information über die zurückgelegte Strecke übertragen werden müssen.

3.2 Bildverarbeitung

Die Bildverarbeitung ist bei einem Roboter eine der wichtigsten Grundfähigkeiten, wenn er sich mit Hilfe von visuellen Informationen orientieren und Entscheidungen anhand von sichtbaren Vorgängen treffen soll.

Der Sensor *Kamera* liefert eine enorme Datenmenge. Zum Einsatz kam die Firewire-Kamera DFW-V500 von Sony [11], welche 30 Bilder pro Sekunde im YUV-Format aufnehmen kann. Ein Pixel dieses Bildes besteht aus 2 Byte, so dass bei einer Auflösung von 640*480 Pixeln und 30 Bildern pro Sekunde 18432642 Bytes/s oder ca. 17.5 MB pro Sekunde verarbeitet werden müssen. Diese Daten enthalten große Mengen von Informationen wie man in Abbildung 3.4 erkennen kann.

Im Vorverarbeitungsschritt werden die Bildinformationen mittels der GNU-Bibliotheken `libdc1394` und `libraw1394` von der Kamera geholt. Die Bibliotheken wurden so geändert, dass nicht mehr auf das zu holende Bild gewartet wird, sondern in regelmäßigen Abständen nachgefragt wird ob ein Bild vorhanden ist, und dieses dann abgeholt wird. Diese Modifikation ermöglicht den Neustart der Kamera, falls sie durch statische Aufladung oder eine kurze Unterbrechung der Kabelverbindung abstürzen sollte. So ist das Programm in der Lage den Grabbing-Prozess selbst neu anzustoßen, wofür bei der Standardbibliothek noch ein Eingriff notwendig war.

Die von der Kamera ausgelesenen Daten sind für das menschliche Auge nicht sichtbar verrauscht, scheinbar gleichfarbige Flächen bestehen bei genauer Betrachtung aus unterschiedlichen Schattierungen. Um Algorithmen zu entwickeln, die aus möglichst vielen Bildern interessante und verlässliche Informationen gewinnen, ist es wichtig die Bilddaten eindeutig und schnell weiterzuverarbeiten.

3.2.1 Bildsegmentierung

Vorgehensweise Die Weiterverarbeitung der Bilddaten unterteilt sich in Segmentierung, zur Klassifizierung der Bilddaten, und der eigentlichen Objekterkennung. Zur Segmentierung lassen sich verschiedene Verfahren wie Kantendetektion, Kontrastbilder und Farbsegmentierung verwenden.

Prinzip Da innerhalb der Robocup-Umgebung die wichtigen Objekte fest definierte Farben haben, entschieden wir uns für eine Farbsegmentierung, d.h. der Einteilung aller betrachteten Pixel in bestimmte Farbklassen (1, 2, 3,. . .). Die Segmentierung findet im RGB-Farbraum statt, was eine Umrechnung der Pixel von YUV 4:2:2 in RGB erfordert. Für jede Farbklasse (in unserem Fall: orange, blau, gelb, grün, weiß und schwarz) wird ein Unterraum im RGB-Farbraum definiert. Da diese Unterräume nicht disjunkt sein müssen, wird zusätzlich eine Priorität vergeben, welche die Reihenfolge der Zugehörigkeitstests bestimmt. Betrachtet man einen Pixel, so wird für diesen überprüft ob seine RGB-Werte im Unterraum einer Farbklasse liegen, dies wird für jede Farbklasse angefangen mit dem Unterraum der höchsten Priorität geprüft bis der Pixel einer Farbklasse zugeordnet werden konnte.Da nicht unbedingt der gesamte RGB-Farbraum durch Farbklassen abgedeckt ist, können möglicherweise einzelne Pixel keiner Farbklasse zugeordnet werden. Solche werden zur einfacheren Verarbeitung in einer Rest-Farbklasse (0) gesammelt.

Umsetzung Eine gängige Methode zur Einteilung des RGB-Raumes in Farben ist die Erstellung von LookUp-Tables, die alle zu einer Farbklasse gehörenden RGB-Werte in Tabellenform enthalten und somit die Zugehörigkeit eines Pixels in einer Laufzeit von O(1) liefern. In einer Umgebung mit stark schwankenden Lichtverhältnissen wie beim Robocup ist die Erstellung von LookUp-Tables sehr aufwendig, da diese oftmals zu viele Werte enthalten, so dass Objekte segmentiert werden, die nicht erkannt werden sollen, oder zu wenig Werte enthalten und die Objekte somit nicht vollständig segmentiert werden. So entsteht zum Beispiel durch die in der Regel zur Beleuchtung verwendeten Scheinwerfer ein starker kegelförmiger Lichteinfall von oben. Dieser sorgt zu einer Entstehung von Glanzeffekten auf der Oberseite des Balls und zu einer Schattenbildung auf der Unterseite, so dass der Ball im Kamerabild nicht nur Orange erscheint, sondern von Weiß bis Dunkelrot alle Schattierungen

durchläuft. Ähnliche Probleme treten bei den Toren auf, da hier von der Latte und den Pfosten Schatten auf die Rückwand des Tors geworfen werden, die gerade beim blauen Tor zu einer fast schwarzen Färbung des Tors im Kamerabild führen. Um diese Probleme zu lösen mussten wir eine Methode verwenden, die es uns ermöglicht, die Farbklassen möglichst einfach und schnell den gerade herrschenden Lichtverhältnissen anzupassen. Die von uns verwandte Methode ermöglicht mittels dynamischer Einstellungen die Definition von Unterräumen der Farbklassen. Für jede Farbklasse können Intervallgrenzen angegeben werden, welche die Position im RGB-Raum bestimmen. Beispielsweise können für die Farbklasse Blau die untere Intervallgrenze des Blauanteils des RGB-Wertes sowie die oberen Intervallgrenzen für den Rot- und den Grünanteil des RGB-Wertes eingestellt werden. D.h. alle Pixel deren RGB-Werte zum Beispiel in den Intervallen Rot (0-62), Grün (0-78) und Blau (145-255) liegen, gehören zu der Farbklasse Blau. Die Einstellung von drei der sechs Intervallgrenzen genügt, um eine eindeutige Zuordnung der Objekte zu ihren Farbklassen zu gewährleisten. In Tabelle 3.2 sieht man die Einstellungsmöglichkeiten der Intervallgrenzen bei den einzelnen Farbklassen.

Farbklasse	Rot-Intervallgrenzen		Grün-Intervallgrenzen		Blau-Intervallgrenzen	
	Untere	Obere	Untere	Obere	Untere	Obere
Orange	Dyn.	255	0	Dyn.	0	Dyn.
Blau	0	Dyn.	0	Dyn.	Dyn.	255
Gelb	Dyn.	255	Dyn.	255	0	Dyn.
Schwarz	0	Dyn.	0	Dyn.	0	Dyn.
Weiß	Dyn.	255	Dyn.	255	Dyn.	255
Grün	0	Dyn.	Dyn.	255	0	Dyn.

Tabelle 3.2: Nach Priorität geordnete Farbklassen mit Angabe der Einstellmöglichkeiten der Intervallgrenzen

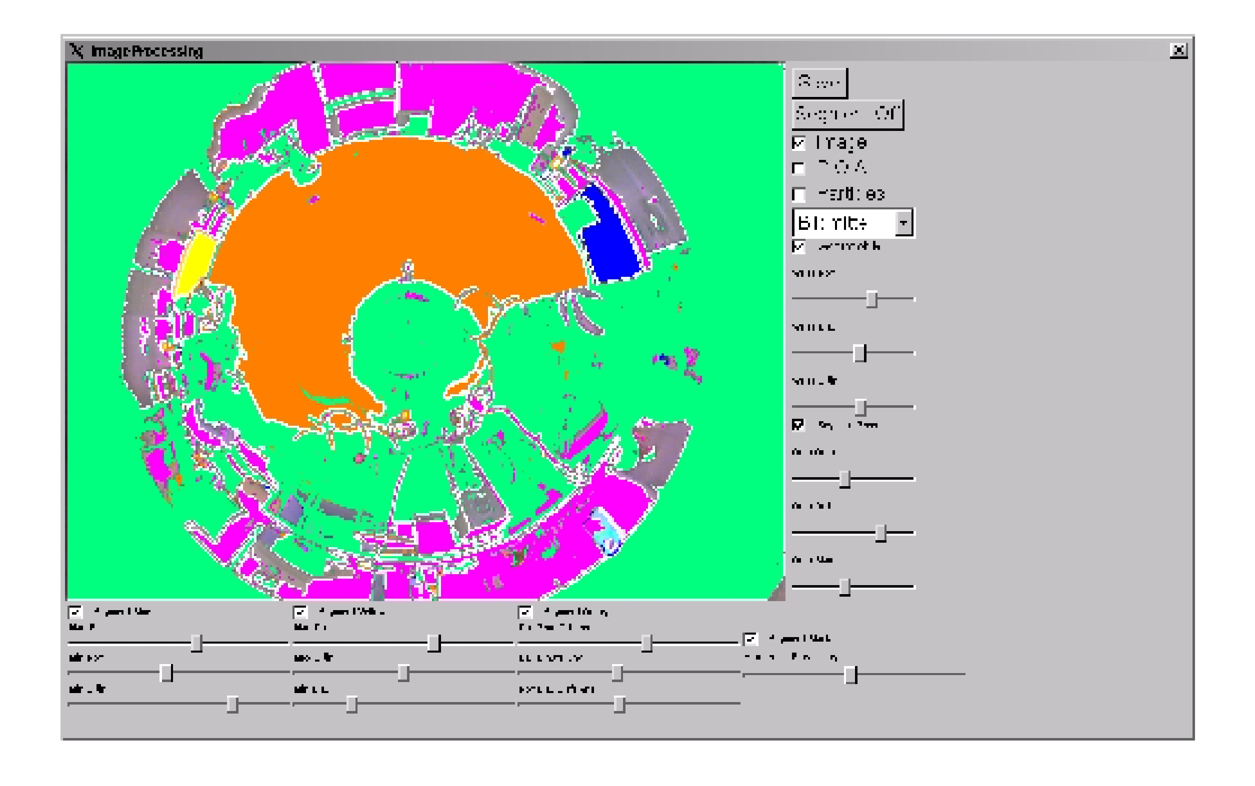


Abbildung 3.4: Bildverarbeitungsfenster mit Segmentierung aller Farbklassen

3.2.2 Objektsuche

Um interessante Objekte im Bild zu finden, ist es sehr hilfreich das Bild in Bereiche einzuteilen und nur dort zu suchen, wo sich das gewünschte Objekt aufhalten kann. Dies bringt Geschwindigkeitsvorteile, denn es müssen nicht alle Pixel im Bild betrachtet werden, da sich bestimmte Objekte nur in bestimmten Bereichen im Bild aufhalten können.

Hierzu muss man sich zuerst den Aufbau eines omnidirektionalen Bildes vom Spielfeld verdeutlichen, um Regeln zu finden wo sich *regions of interest* für einzelne Objekte befinden.

Mittig sitzt das Kameraobjektiv. Wenn ein Spiegel mit einer sehr stark gewölbten Mitte oder einer Spitze verwendet wird, dann kann das Objektiv sogar verschwinden. Dies ist wünschenswert, denn so stehen mehr Pixel im Bild für interessante Flächen zur Verfügung. Nach dem Kameraobjektiv schneidet die Sichtlinie der Kamera direkt den Boden, der untere Rand des Roboters wird nicht wahrgenommen (hier entsteht ein toter Winkel), der aber zu klein ist, um den Ball ganz zu verdecken (vgl. Abbildung 3.5). Die Entfernung des Schnittpunktes mit dem Boden nimmt bei steigendem Pixelabstand im Bild immer stärker zu, bis sie am Horizont die Unendlichkeit erreicht (die Sichtlinie verläuft dann parallel zum Boden). In diesem Bildbereich liegen alle Objekte, die sich auf der gleichen Höhe der Kamera befinden.

Tore und Eckfahnen Die Horizontlinie schneidet die wichtigen Objekte *Tor* und *Eckfahne* distanzunabhängig von jedem Punkt auf dem Feld. Somit ist es sinnvoll die Suche auf den Horizontbereich zu konzentrieren (vgl. Abbildung 3.5).

Um die ersten Pixel zu finden, die jeweils zu einem Tor oder einer Eckfahne gehören, wird im Bereich der Horizontlinie eine randomisierte Suche gestartet. In jedem Durchlauf werden 100 Pixel in einem ringförmigen Bereich darum durch die Funktion `seedSearch()` (siehe auch Algorithmus 2) auf verschiedene Bedingungen untersucht. Ziel dieser Funktion ist die eindeutige Zuordnung möglichst vieler Pixel zu den Objekten *Tor Gelb*,*Tor Blau*,*Pole Gelb Links*, *Pole Gelb Rechts*, *Pole Blau Links* und *Pole Blau Rechts*.
Ist ein Pixel als blau erkannt, so wird zuerst angenommen, dass er zum blauen Tor gehöre. Auf diesem Pixel wird daraufhin die Funktion `torBlauTest(Pixel)` durchgeführt. Sie testet, ob in Richtung des Eingabepixels auch die Farbe Gelb erkannt wurde. Ist dies der Fall, liegt offensichtlich eine Eckfahne vor. Je nach Abfolge der Erkennung Blau/Gelb/Blau oder Gelb/Blau/Gelb wird der Pixel einem Eckfahnen-Array zugeordnet. Die Reihenfolge der Eckfahnen wird je nach Lage der Tore bestimmt.

Ball Der Ball kann sich in einem omnidirektionalen Bild lediglich zwischen dem äußeren Rand des Kameraobjektivs und der maximalen Ausdehnung des Feldes im Bild aufhalten. Durch die Suche in nur dieser Region verhindert man, dass Objekte, die sich außerhalb des Spielfelds befinden wie etwa Kleidungsstücke, fälschlicherweise als Ball erkannt werden. Dies ist jedoch nur bedingt möglich, da sich z.B. Schuhe durchaus in diesem Bereich aufhalten können.

Hindernisse Die Hindernisse, also Roboter, sind am leichtesten durch ihre schwarze Farbe zu erkennen. Ihr Aufenthaltsbereich entspricht in etwa dem des Balls. Allerdings sind die schwarzen Hindernisse meistens so hoch, dass sie sich über große Teile des Bilder erstrecken und dabei Landmarken oder den Ball verdecken können. Da es immer mehrere Hindernisse gibt, benutzen wir zum Erkennen und Verfolgen von Hindernissen einen etwas anderen Ansatz: Hindernisse werden mit Hilfe von ca. 50 Scanlinien erkannt, die radial vom Roboterradius in Richtung Feldgrenze verlaufen. Wird auf einer dieser Scanlinien ein Übergang von der Feldfarbe Grün/Weiß zu Schwarz festgestellt, so wird in diesem Winkel ein Hindernis weitergegeben. Die Entfernung des Hindernisses lässt sich hierbei aus der durch die Distanzfunktion umgerechneten Pixelentfernung errechnen, da der Übergang Feld/Hindernis am Boden stattfindet. Bei dem Algorithmus, der den Übergang feststellt, werden Schwellwerte eingesetzt, um die schwarzen Marker auf dem Boden, Bildstörungen oder Schatten des Balls nicht als Hindernis erscheinen zu lassen.

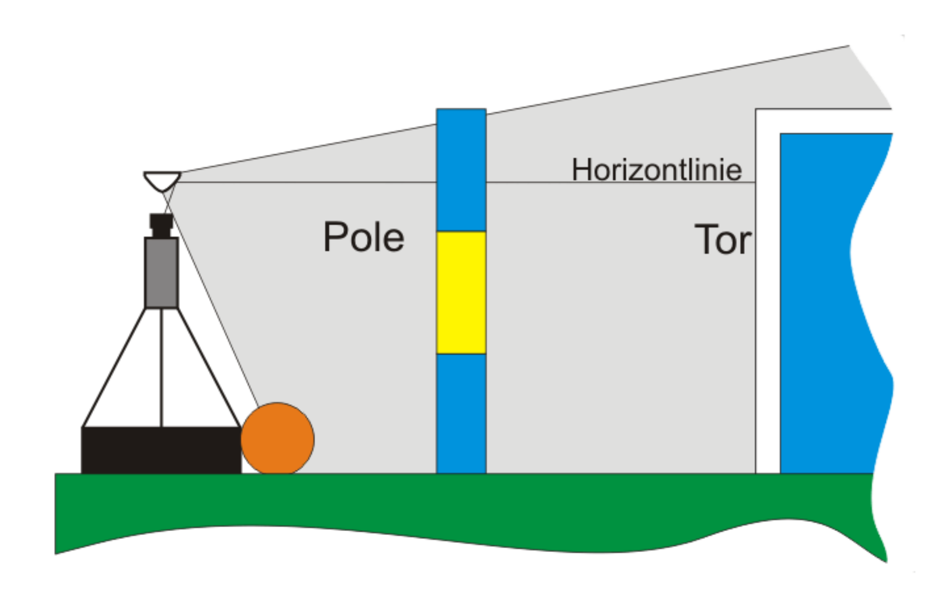


Abbildung 3.5: Schema des sichtbaren Bereichs im Spiegel (grau)

Objekt	Größe im Bild		Maximale Entfernung	Anzahl möglicher Objekte
	nah	fern		
Ball	groß	verschwindend	ca. 4-5 m	1
Tore	sehr groß	mittel	mehr als 10 m	2
Poles	groß	sehr klein	ca. 8 m	4
Hindernisse	groß	sehr klein	ca. 5-6 m	8 und mehr

Tabelle 3.3: Charakteristische Objekteigenschaften im Kamerabild

Bei dieser Vorgehensweise kommt es durchaus vor, dass ein Hindernis von mehreren Scanlinien erfasst wird. Eine eindeutige Trennung von verschiedenen Hindernissen ist hierdurch nicht möglich, doch ist diese bei nebeneinanderstehenden Robotern im Kamerabild auch für das menschliche Auge nur schwer möglich.

Distanz und Winkelbestimmung Es ist klar, dass die eingezeichneten Begrenzungskreise mit der Bildgeometrie nicht genau übereinstimmen. Das wird durch die vielen Parameter des Spiegel und der Kamerahalterung beeinflusst. Sind diese beiden Elemente perfekt aufeinander ausgerichtet und diese Kombination wiederum parallel zur Feldoberfläche, so werden Winkel zu Objekten korrekt abgebildet. Senkrechte Kanten bilden im omnidirektionalen Bild radiale Linien und Kreise um den Roboter ergeben auch wieder Kreise im Bild. Diese Eigenschaft bedeutet, dass gleiche Entfernungen in jeder Richtung auf die gleiche Entfernung von der Bildmitte abgebildet werden. Da dies unter den gegeben technischen Voraussetzungen nur bedingt einstellbar ist, ist es mitunter schwierig Pixeldistanzen direkt auf reale Distanz umzurechnen. Winkel dagegen werden einigermassen, wenn auch nicht völlig, gleichmäßig abgebildet (vgl. Abschnitt 3.2.3).

3.2.3 Distanzfunktion

Durch die technischen Schwierigkeiten bei der korrekten Ausrichtung von Kamera auf Spiegel und dieser Einheit auf die Umgebung, ist es nicht möglich ein Bild zu erzeugen in dem die Verzerrung

des Spiegels in alle Richtungen gleichförmig ist. Dadurch werden vor allem Distanzmessungen stark beeinflusst, welche aber für Teile des Entscheidungsablaufs im Programm wie die Ballanfahrt und die *Obstacle Avoidance* essentiell wichtig sind, so dass notwendigerweise diese Messungen bezüglich der aktuellen Spiegeleinstellungen umgerechnet und korrigiert werden. Zur Aufstellung der Distanzfunktion wurde eine Messtechnik verwendet, die die Realdistanz zu jedem auf dem Boden befindlichen Punkt im Bild berechnen kann.

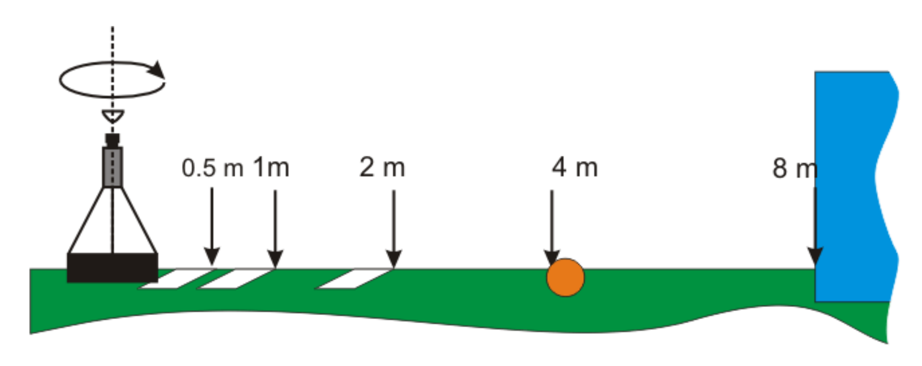


Abbildung 3.6: Schema der Umgebung für eine Distanzmessung

Kalibrierung Zur Messung ist eine geeignete Umgebung nötig in der festgelegte Marker möglichst verwechslungsfrei erkannt werden. Abbildung 3.6 beschreibt die benutzte Umgebung: sie besteht aus dem Ball, drei weißen DIN-A4-Seiten und dem blauen Tor jeweils in definierten Entfernungen. Dem Distanzmessungsalgorithmus dient der Ball als Referenzpunkt, er kann in jedem Bild zweifelsfrei erkannt werden. Der Algorithmus sammelt folgende Farbübergänge als Messdaten:

- Ball: Übergang des Balls zum Boden
- Marker: Übergänge von Markern zum Feld
- Tor: Übergang vom Feld zum Tor

Wurde der Ball gefunden und der Distanzmessungsalgorithmus in der GUI gestartet, beginnt der Roboter eine langsame Drehbewegung.

Distanztabelle Der Algorithmus sammelt die Pixelabstände der Messpunkte zu jedem Zeitpunkt und trägt sie zusammen mit dem zum Ball gemessenen Winkel in ein Array von `MarkerFunctions` ein. Durch die Methode `add_measure()` wird eine fortschreitende Mittelung von mehreren Messwerten in gleicher Richtung vorgenommen. Sobald der Algorithmus mindestens fünf Messwerte aus allen 360 Richtungen genommen hat, stoppt er die Drehung. Danach werden die Messwerte des Arrays von `MarkerFunctions` mit einem einfachen Tiefpaß-Filter geglättet. Hierdurch ergibt sich eine Charakteristik eines Kamera/Spiegelaufbaus, die durch die fünf Messwerte in jede Richtung definiert ist. Danach werden Distanzen mit Hilfe linearer Interpolation zwischen den nun bekannten Entfernungen der Stützpunkte bestimmt.

Ein Messergebnis der Abstände der Marker zum Roboter ist in Abbildung 3.7 erkennbar. Wäre die Kameraanordnung perfekt eingestellt, so würden sich Geraden ergeben.

3.2.4 Objekttracking

Zur Zeiteinsparung bei der Bildverarbeitung nutzt man bestehendes Vorwissen über die Position von Objekten, sie sind so im aktuellen Bild schneller zu finden. Dieses als *Tracking* bekannte Prinzip

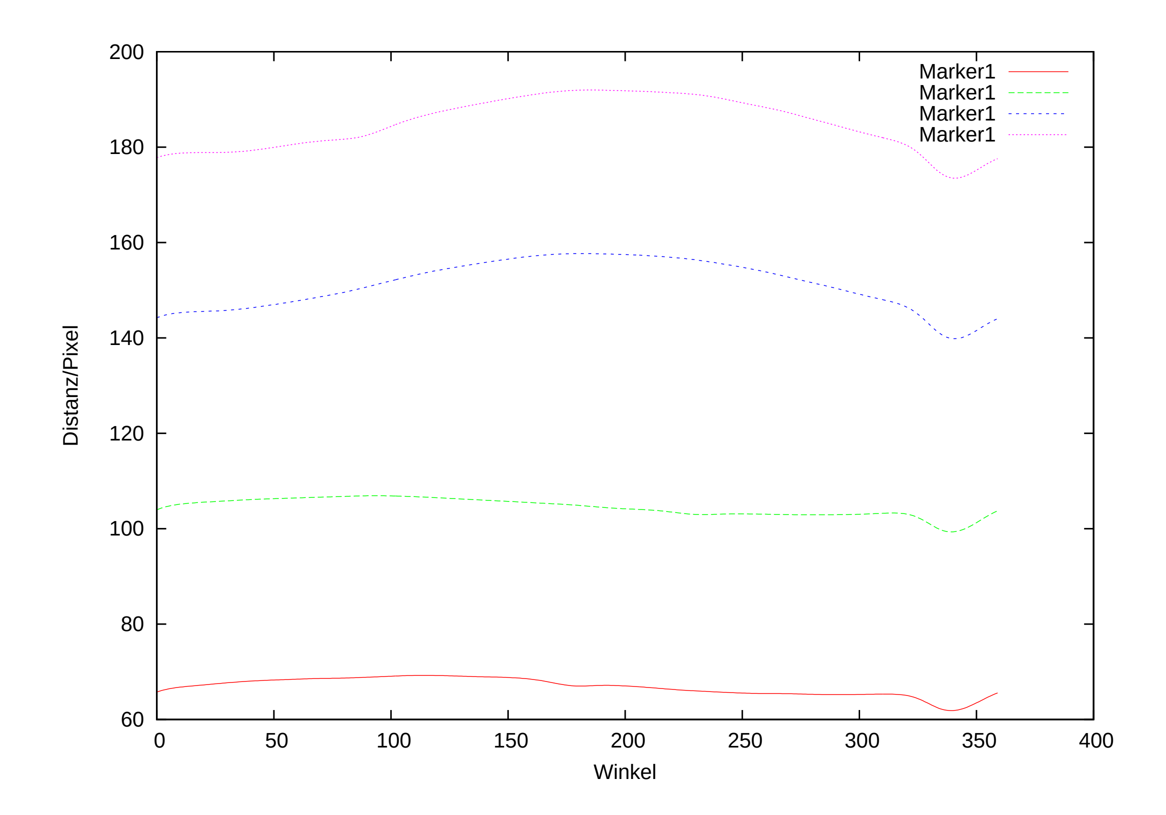


Abbildung 3.7: Aufgenommene Daten der Distanzkalibrierung. Die einzelnen Linien entsprechen dem Pixelabstand der verschiedenen Marker von der Bildmitte bei der Distanzkalibrierung. Anhand des Kurvenverlaufs lässt sich erkennen, dass Spiegel und Kamera nicht optimal aufeinander ausgerichtet sind.

unterstützt eine sehr genaue Objekterkennung über eine längere Bildfolge. Die erste Positionsbestimmung eines Objektes in einem Bild kann durchaus noch ungenau sein, wenn diese Position über mehrere Bilder verfeinert wird. Dieses Verfahren eignet sich besonders für die Verfolgung kleinerer Objekte in der Entfernung, dabei beträgt beispielsweise die Ballgröße ab etwa fünf Meter nicht viel mehr als 2 - 3 Pixel.

Als Ausgangspunkt wird in jedem Schritt entweder das Ergebnis des letzten Schleifendurchlaufes oder die grobe Aufteilung nach dem Durchlauf des `seedSearch()`-Algorithmus (siehe Algorithmus 2) benutzt. Bei den Toren werden Pixel, die in der Suchphase dem Tor zugeordnet wurden, fortgepflanzt, indem je 500 Nachfolger aus den gefunden Pixeln erzeugt werden. Die Nachfolger befinden sich bei den Toren und Eckfahnen direkt radial versetzt um +/- 20 °. Beim Ball werden sie um +/- 20 Pixel horizontal und vertikal versetzt gewählt.

Die Idee des Algorithmus besteht darin, dass wirklich zum Tor gehörende Pixel bzw. dem zu verfolgenden Objekt gehörende besonders viele Nachbarn haben, die die gleiche Bedingung erfüllen. Deswegen überleben diesen Fortpflanzungsschritt hauptsächlich Pixel, die in den größten zusammenhängenden Farbflächen wie *Tor* und *Ball* zu finden sind. Die Mittelpunkte dieser Objekte werden durch Mittelung der zugehörigen Pixelpositionen berechnet.

Durch die gewonnene Zeit bei der Suche können im Programm, dessen Laufzeit sich dadurch ebenfalls verkürzt, mehr Bilder bearbeitet werden: es ergeben sich schnellere Bildfolgen. So werden Bewegungen im Bild nur durch kleine Änderungen von Bild zu Bild sichtbar. Schnelle Objekte wie der Ball werden in jedem Bild allein durch den Fortpflanzungsschritt und durch die Information aus dem letzten Bild erfasst und so effektiv verfolgt werden , ohne einen neuen Suchschritt zu benötigen.

Eine schematische Implementierung des Tracking-Algorithmus wie er für den Ball, die Tore und

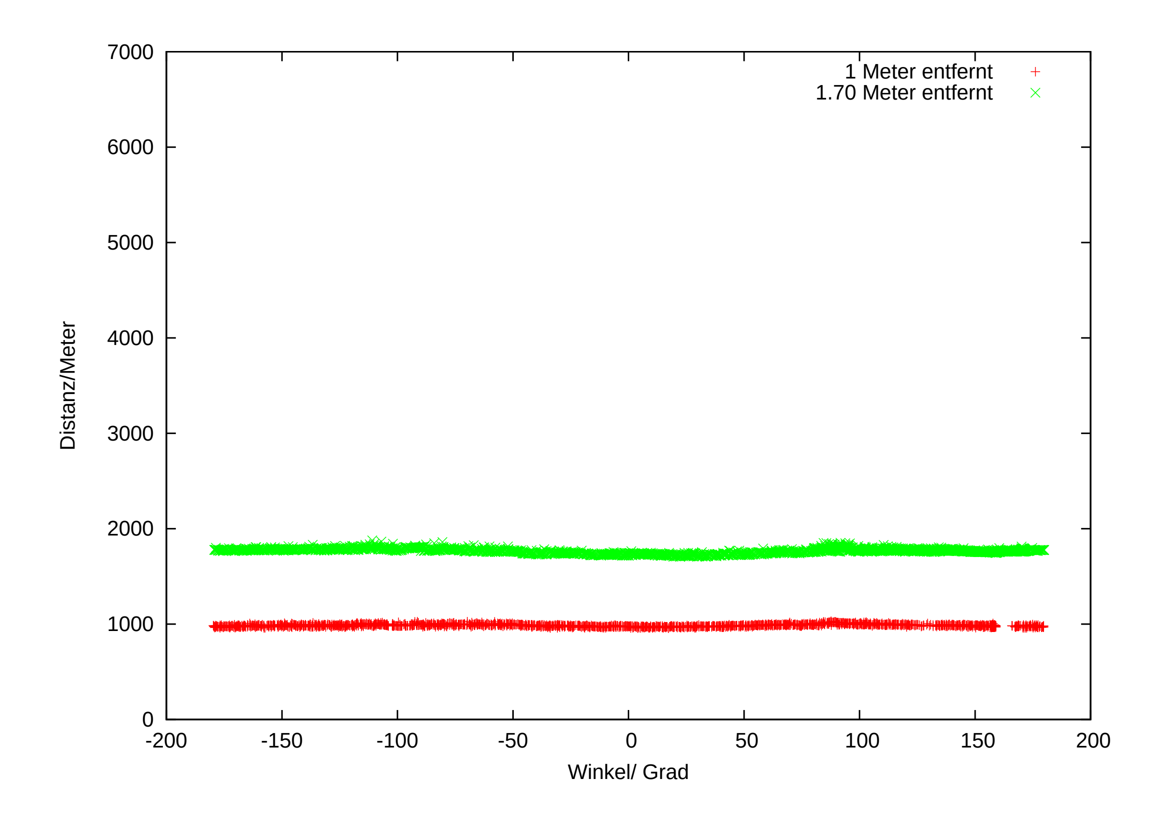


Abbildung 3.8: Ergebnisse einer Distanzmessung nach der Distanzkalibrierung

die Eckfahnen eingesetzt wurde besteht aus den beiden Teil-Algorithmen `sample_step()` (vgl. Algorithmus 3) und `propagate_step()` (vgl. Algorithmus 4)[8].

Beim `sample_step`-Algorithmus wird eine festgelegte Pixelanzahl *AnzahlBallPartikel* im Suchraum auf eine Farbe hin untersucht. Die zu untersuchenden Pixel mit Abstand r und Winkel φ vom Bildmittelpunkt werden dabei zufällig gezogen. Beim Ball ist der Suchraum der ringförmige Bereich im Bild zwischen dem Roboterradius $R_{Roboter}$ und dem Spielfeldradius R_{Feld}. Alle diesem Test gegenüber positiven Pixel werden in das Array `PositiveBallPartikel` einsortiert.

Im `propagate_step`-Algorithmus werden aus den bereits gefundenen positiv markierten Ballpartikeln, die sich im Array `PositiveBallPartikel` befinden, `AnzahlBallPartikel` Testkandidaten ausgewählt. Die Testkanditaten werden gleichverteilt aus dem Array `PositiveBallPartikel` gezogen, und um +/- 20 Pixel in x- und y-Richtung verschoben. Sind diese wiederum in der Farbklasse Ball enthalten, so werden sie in das Array `PositiveBallPartikel` übernommen. Wenn sich nach der Suche mindestens ein Partikel in dem Array befindet, so gilt der Ball als gefunden. Der Mittelwert der positiven Ballpartikel wird berechnet und anhand des Partikels mit dem kleinsten Abstand zur Bildmitte skaliert. So ergibt sich eine recht genaue Abschätzung des Aufsatzpunkts des Balls. Dieser Aufsatzpunkt wird daraufhin mit Hilfe der Distanzfunktion in eine reale Entfernung umgerechnet und in das PossibleObject-Array zur Weiterverabeitung übernommen.

[8]Dargestellt ist in beiden Fällen der Ballalgorithmus.

```
1:  for i = 0 to 100 do
2:     r = rand (R_Horizont1 ... R_Horizont2) {R_Horizont1 und R_Horizont2 begrenzen den ungefähren Be-
       reich des Bildes, wo der Horizont zu finden ist.}
3:     φ = rand (−π ... π)
4:     Bestimme x, y aus r, φ mit Bildmitte (mitte_x, mitte_y)
5:     if getPixelColor(x,y)==BLUE then
6:        if poleBlueTest(x,y)==TRUE then
7:           Addiere Partikel (x, y) zu PositvePoleBluePartikel[]
8:        end if
9:        if goalBlueTest(x,y)==TRUE then
10:          Addiere Partikel (x, y) zu PositveGoalBluePartikel[]
11:       end if
12:    end if
13:    if getPixelColor(x,y)==YELLOW then
14:       if poleYellowTest(x,y)==TRUE then
15:          Addiere Partikel (x, y) zu PositvePoleYellowPartikel[]
16:       end if
17:       if goalYellowTest(x,y)==TRUE then
18:          Addiere Partikel (x, y) zu PositveGoalYellowPartikel[]
19:       end if
20:    end if
21: end for
```

Algorithmus 2: *seed_search*() Funktion, die blaue und gelbe Pixel den Toren und Poles zuordnen kann.

```
1: for i = 0 to AnzahlBallPartikel do
2:    r = rand (R_Roboter ... R_Feld)
3:    φ = rand (−π ... π)
4:    Bestimme x, y aus r, φ mit Bildmitte (mitte_x, mitte_y)
5:    if getPixelColor(x,y)==BALL then
6:       Addiere Partikel (x, y) zu PositveBallPartikel[]
7:    end if
8: end for
```

Algorithmus 3: *sample_step*() Funktion, die dem Objekte zugehörige Pixel finden soll.

```
1: for i = 0 to AnzahlBallPartikel do
2:    z = rand (0 ... PositiveBallPartikel.size()) {Bestimme zufällig zu untersuchenden Partikel}
3:    x_Blur, y_Blur = rand (−20 ... + 20) {Zufällige Wahl eines Punktes in der Partikelumgebung}
4:    Erzeuge neuen Partikel (x_Test, y_Test)
      mit x_Test = PositiveBallPartikel[z].x + x_Blur
      und y_Test = PositiveBallPartikel[z].y + y_Blur
5:    if getPixelColor(x_test,y_test)==BALL then
6:       Füge Partikel (x_test, y_test) zu PositveBallPartikel[] hinzu
7:    end if
8: end for
```

Algorithmus 4: *propagate_step*() Funktion, die gefunden Partikel über Zeit propagiert.

3.3 Weltmodell

Die Wahrnehmung der Umwelt hat unmittelbaren Einfluss auf unsere Handlungen, visuelle Informationen werden im menschlichen Gehirn zu Objekten und assoziierten Aktionen umgesetzt. Je präziser die gewonnenen Informationen sind, desto besser ist die Aktionsauswahl möglich.

Der Roboter an sich verfügt über keine eigenständige Wahrnehmung. Abhängig von seinem Einsatzgebiet wird entschieden, welche Größen in seiner Umgebung relevant sind und wie er darauf reagieren soll. Trägt man die erkannten Objekte in eine Umgebungskarte ein, die den näheren Bereich des Roboters beschreibt, repräsentiert sie ein Modell der Welt, wie sie für den Roboter zu sein scheint. Das Weltmodell soll dabei eine sehr detaillierte, in Echtzeit berechenbare Darstellung sein, um dem Strategiemodul eine optimale Grundlage für eine Entscheidung zu bieten.

3.3.1 Repräsentation der Umwelt

Im Rahmen der Projektgruppe entsprach die Roboterumgebung (vgl. [1]) einem Fußballfeld mit

1. weißen Spielfeldmarkierungen
2. jeweils einem blauen sowie einem gelben Tor
3. vier Eckfahnen (Poles) (2 blau-gelb-blau und zwei gelb-blau-gelb)
4. einem orange-farbenen Ball
5. diversen schwarzen Hindernissen

Genau diese Aspekte müssen in das Weltmodell des Roboters einfließen, das durch die Klasse `CWorldModel` modelliert wird. Sie erhält ihre Eingabedaten entweder aus der Bildverarbeitung oder aus dem Simulator `CDortmundClient`, oder aus aufgezeichneten Daten.

Diese Daten sind repräsentiert durch die Klasse `CPossibleObject`, eine erweiterbare Containerklasse für erkannte Objekte aus der Bildverarbeitung. Die Typen der verschiedenen möglichen Objekte sind in der Datei `Global_Defines.h` definiert. Darunter finden sich

- Ball: der Ball im Roboterkoordinatensystem
- Tore: Richtung und Entfernung der Tore
- Torpfosten: Richtung der Torpfosten
- Eckfahne: Richtung der Eckfahne (Pole)
- Hindernisse: Richtung und Entfernung eines Hindernisses

3.3.2 Selbstlokalisation

Für das Entscheidungsmodul und dessen spätere Aktionsauswahl ist es besonders wichtig zu wissen, wo sich der Roboter auf dem Spielfeld befindet. So sind viele Verhaltensweisen eines Roboters beim Robocup von dessen aktueller Position auf dem Spielfeld abhängig, z.B. die richtige Positionierung des Torwarts oder der Feldspieler, der Torschuss oder das Teamplay.

Lokale Lokalisation Man kann versuchen, die Position lediglich aus einer bekannten Startposition und durch Messen des zurückgelegten Weges zu berechnen (*dead reckoning*). Aufgrund der Dynamik und der Dauer des Spiels ist diese Methode nicht ausreichend. Fehler durch Kollisionen oder Schlupf der Räder werden hierbei aufsummiert und führen zu immer größer werdenden Abweichungen der wahrgenommenen von der tatsächlichen Position. In diesem Zusammenhang wird auch das Kidnapped-Robot-Problem [4] beobachtet: ein Roboter wird von Hand umgesetzt, ohne dass seine

Sensoren dies feststellen können. Dies führt zum totalen Positionsverlust, der durch lokale Lokalisation nicht wahrgenommen werden kann. Ein Roboter, der das Kidnapped-Robot-Problem lösen kann, muss diese Ortsänderung erkennen und sich global neu lokalisieren.

Globale Lokalisation Zur Feststellung der aktuellen Roboterposition ist es sinnvoll, die Position von fest definierten Landmarken der Robocup-Umgebung zu erkennen und als Orientierungshilfe zu nutzen. Hierzu zählen beispielsweise die Tore, die Eckfahnen und die weißen Linien. Es ist möglich die Position des Roboters mit Hilfe von Triangulation direkt zu berechnen, doch aufgrund der teilweise sehr ungenauen oder falschen Bildverarbeitungsinformation ist es schwierig, die dafür nötigen Landmarken sinnvoll auszuwählen. Deswegen wurde ein Monte-Carlo Ansatz zur Fusion der Sensorinformation eingesetzt, der auch verrauschte Informationen verarbeiten kann.

Sensoren Der Beobachtungssensor zur *globalen Lokalisation* ist der omnidirektionale Kameraaufbau. Die Landmarken sind aufgrund ihrer Farbkodierung und ihrer definierten Positionen am leichtesten durch eine Kamera zu erkennen. Als Aktionssensor zur *lokalen Lokalisation* dient die Odometrieinformation, welche von der Roboterschnittstelle bereitgestellt wird. Aufgrund der Charakteristik der Sensoren erzeugt die Kamera Daten mit hohem Rauschanteil, die über die Zeit gleichbleibend genau sind. Der Aktionssensor, also die Odometrie, erzeugt sehr gering verrauschte Daten, die allerdings nur über kurze Zeit genau sind und deren Fehler sich schnell akkumulieren.

Verwendete Sensorinformationen Zur Selbstlokalisation werden von der Bildverarbeitung folgende Sensorinformationen bereitgestellt:

- 2 Tore: Die Winkel zu den Mittelpunkten der Tore
- 4 Torpfosten: Der Rechte und linke Torpfosten des Tors
- 4 Torpfosten-Fußpunkte: der Fußpunkt eines Torpfosten
- 4 Eckfahnen

Die Sensorinformationen sind nicht immer vollständig, da einige Landmarken aufgrund des Rauschens oder weil sie von Hindernissen verdeckt werden nicht stabil erkannt werden können.

Als Haupteingabedaten für den Selbstlokalisationsalgorithmus erwiesen sich die Torpfosten als besonders stabil und zuverlässig. Die Winkel zu den Kanten der Tore sind fast von jeder Position des Feldes aus, oft auch über Hindernisse hinweg und innerhalb des Tors, sichtbar. Generell werden beim gewählten Kameraaufbau die Winkel zu Objekten recht genau auch über größere Distanzen hinweg wiedergegeben. Distanzen sind im Bereich von 0-4 m recht gut, darüber hinaus nur schlecht messbar. Die Messung größerer Distanzen kann durch Robotervibrationen und durch einen unpräzisen Kameraaufbau während des Fahrens stark schwanken. Der Torwart erkennt die Fußpunkte der Torpfosten sowie deren Distanzen gut. Sie werden für die Selbstlokalisation vor dem Tor und im Tor benutzt.

Die einzelnen Sensorinformationen der Bildverarbeitung und der Odometriesensoren werden durch einen Monte Carlo Particle Filtering Ansatz fusioniert. So kann die Roboterposition und -ausrichtung global bestimmt, und lokal verfolgt werden. [2, 12].

Monte Carlo Particle Filtering

Das Grundprinzip des Verfahrens liegt darin, das Modell der Aufenthaltswahrscheinlichkeit durch Sensordaten und Odometriedaten zu aktualisieren. Ziel ist die Zentrierung der Aufenthaltswahrscheinlichkeit um die aktuelle Roboterposition, wobei die entsprechende Wahrscheinlichkeitsdichte durch ein Sampleset von Partikeln repräsentiert wird.

Die Wahrscheinlichkeitsdichte $Bel\,(l_t \mid d_{t\ldots0})$ repräsentiert die Position l_t des Roboters, also seine x- und y- Koordinate und seine Ausrichtung φ zum Zeitpunkt t wenn alle Daten d bekannt sind. Die Daten bestehen aus Odometrieinformationen a und Sensoreingaben s. Es gilt:

$$Bel\,(l_t) = p\,(l_t \mid s_t, a_{t-1}, s_{t-1}, a_{t-2}, \ldots, s_0) \tag{3.1}$$

Durch Anwendung der Bayes'schen Regel und Berücksichtigung eines Markovprozesses erster Ordnung lässt sich die Rekursionsgleichung

$$Bel\,(l_t) = \alpha \cdot p\,(s_t \mid l_t) \int p\,(l_t \mid l_{t-1}, a_{t-1}) \cdot Bel\,(l_t \mid d_{t-1}) \mathrm{d}l_{t-1} \tag{3.2}$$

ableiten, wobei der Faktor α zur Normalisierung der Wahrscheinlichkeiten dient.

Odometrie Bei der Aktualisierung durch Odometriedaten wird die Roboterbewegung durch die bedingte Wahrscheinlichkeit $p\,(l_t \mid l_{t-1}, a_{t-1})$ dargestellt, also die Wahrscheinlichkeit , dass der Roboter von der Position l_{t-1} durch die Bewegung a_{t-1} an die Position l_t gelangt. Diese Wahrscheinlichkeit modelliert die Messungenauigkeiten der Odometrie, hierzu zählt z.B. der Schlupf der Räder. Je genauer die Odometrie bzw. je besser man sich auf die Odometriedaten verlassen kann, desto geringer wählt man die Varianz der Gaußverteilung. Die Aktualisierung geschieht mit der Gleichung

$$Bel\,(l_t) \leftarrow \int p\,(l_t \mid l_{t-1}, a_{t-1}) \cdot Bel\,(l_t \mid d_{t-1}) \mathrm{d}l_{t-1} \tag{3.3}$$

Validierung Bei der Validierung werden die Partikelwahrscheinlichkeiten durch die Sensordaten neu gewichtet:

$$Bel\,(l_t) \leftarrow \alpha p\,(s_t \mid l_t) \tag{3.4}$$

Es wird die Wahrscheinlichkeit berechnet, dass für einen Partikel der Verteilung eine Sensormessung möglich ist. Dabei sind beliebige Sensordaten kombinierbar, z.B. die Winkel unter denen Landmarken gesehen werden und Distanzen zu Landmarken. Dieses zweistufige Verfahren kann das globale Lokalisierungsproblem lösen, eine Vorgabe der Startposition ist überflüssig. Das Kidnapped-Robot-Problem wird von diesem Verfahren gelöst, da eine Validierung der Position zu einer sinkenden Aufenthaltswahrscheinlichkeit führt, die dadurch zu einer Gleichverteilung konvergiert.

Sampling/Importance Resampling Aus den Partikeln werden nun N neue Partikel erzeugt. Hierbei werden die Partikel mit höheren Wahrscheinlichkeit häufiger ausgewählt. Diese entsprechen der Position des Roboters am besten, und nur in diese interessanten Partikel wird weitere Rechenzeit investiert.

Implementierung des Algorithmus

Der komplette Algorithmus ist in der Klasse `CHohensyburg`[9] umgesetzt.

Die Anzahl der zur Repräsentierung der Verteilung nötigen Partikel ist so gewählt, dass die Anzahl groß genug für eine stabile Verteilung ist und trotzdem noch klein genug, um den Rechner nicht zu stark zu belasten, in der Praxis haben sich ca. 500 - 700 Partikel als sinnvoll erwiesen. Sie belasten die Durchlaufzeit der Hauptschleife ungefähr mit 6-10 ms pro Durchlauf.

Um die Partikelposition möglichst schnell herauszufinden, werden in der Selbstlokalisation Winkel eingesetzt, die von der Ausrichtung unabhängig sind, z.B. die Winkel *zwischen* Landmarken. Die tatsächliche Ausrichtung des Roboters wird nach Herausfinden der Position direkt durch die Winkel der Landmarken im Bild bestimmt.

[9]Hohensyburg: In Anlehnung an die Kasinos in Monte Carlo wurde das lokal bekannte Kasino Hohensyburg bei Dortmund als Klassenname gewählt.

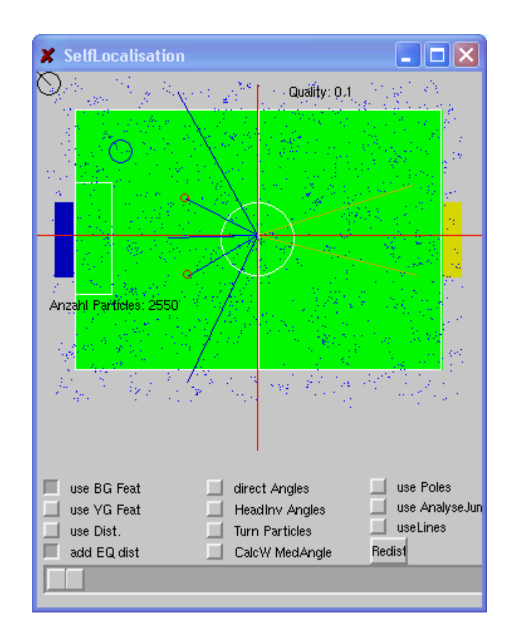


Abbildung 3.9: Gleichverteilung der Partikel zu Beginn des Algorithmus

Um das Kidnapped-Robot Problem zu lösen, beobachtet man ob die Partikelwahrscheinlichkeiten sehr gering sind, um dann eine neue Gleichverteilung durchzuführen, alternativ wird zusätzlich einen Teil gleichverteilter Partikel eingestreut, um eine eventuell bessere Position zu finden.

Zur Partikelrepräsentation dient die Klasse `CParticle`, die einen zweidimensionalen Vektor als Position, einen Winkel als Ausrichtung und eine Fließkomma-Zahl als Gewicht/Wahrscheinlichkeit enthält. Die Klasse `CHohensyburg` enthält ein Array aus Instanzen der Klasse `CParticle`, um die Verteilung $Bel\left(l_t \mid d_{t \ldots 0}\right)$ zu repräsentieren.

Im Folgenden werden die einzelnen Methoden der Klasse `CHohensyburg` vorgestellt:

- `run()` ein Schleifendurchlauf des Partikelfilters
- `equalDistributeParticles()` Gleichverteilung der Partikel über den gesamten Feldbereich (siehe Abbildung 3.9)
- `moveParticlesByOdometry()` Verschieben und Verrauschen der Partikel um die Odometriedaten (vgl. Algorithmus 5)
- `validateParticlesBySensors()` Neugewichtung der Partikel durch Vergleich Sensorwert / erwarteter Wert (vgl. Algorithmus 6)
- `resampleParticles()` Neuziehung der Partikel gemäß ihrer Wahrscheinlichkeit

Als Ausgabe des Algorithmus kann der Partikel mit der größten Wahrscheinlichkeit oder das gewichtete Mittel der Partikelverteilung benutzt werden, beides wird in der Methode `validateParticlesBySensors()` berechnet.

Als Beispiel wird die Implementierung der Methoden `moveParticlesByOdometry()` und `validateParticlesBySensors()` in Pseudocode aufgeführt.

Die Auswirkung dieser verrauschten Verschiebung auf eine Wolke von Partikeln lässt sich auch in der Schemazeichnung (siehe Abbildung 3.11) erkennen.

[10] probability-density-function, Wahrscheinlichkeitsdichtefunktion

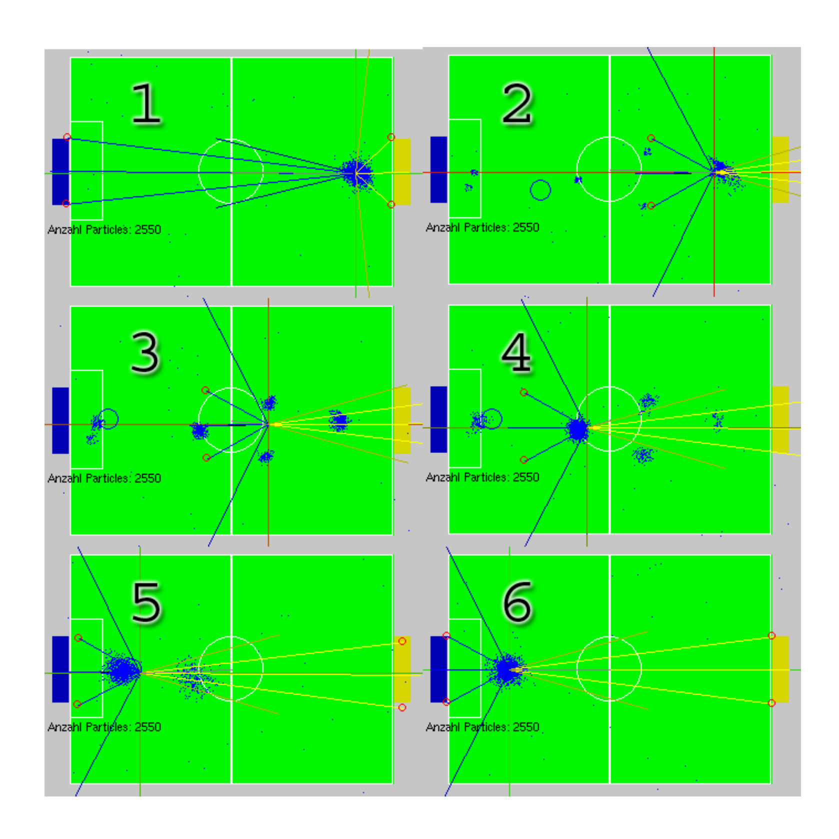


Abbildung 3.10: Demonstration eines Kidnap-Vorganges. Der Roboter, der in Bild 1 rechts vor dem Tor steht, wird umgesetzt. Durch die zusätzlich mit eingestreuten Partikel werden schnell neue Zentren gefunden und die Verteilung verschiebt sich an die neue Position.



Abbildung 3.11: Schema der Verschiebung von Partikeln durch Odometrieinformation. Sequentielle Ausführung einer Verschiebung entlang der x-Achse.

Resampling Nach der Partikelneugewichtung werden die Partikel gemäß ihrer Wahrscheinlichkeit neu gezogen (*Resampling*). Wahrscheinlichere Partikel werden öfter gezogen und unwahrscheinliche Partikel sterben aus, dadurch geht das "Vorwissen" des Algorithmus (die Verteilung) mit in die Positionsberechnung ein. Ein einfacher Algorithmus für die Auswahl läuft in $O(N \log(N))$, es ist jedoch möglich diese Auswahl in $O(N)$ zu treffen[3].

Als Beispiel für das Resampling zeigt die Abbildung 3.12 zeigt die Partikelverteilung einen Schritt nach der Gleichverteilung, bei bekannter Distanz zu den Fußpunkten des blauen Tors. Es lässt sich erkennen, dass sich die Partikel durchsetzen, die in der Entfernung zum Tor mit den gemessenen Werten möglichst gut übereinstimmen.

1: **for** $i = 0$ to $i < m_NumberOfParticles$ **do**
2: $x, y, \varphi = Gauss$ {es werden 3 Gauß-verrauschte Variablen erzeugt}
3: $m_Particles\,[i] + = m_particleOdometry +$ new $CParticle\,(x, y, \varphi)$
4: $i + +$
5: **end for**

Die Odometrieinformation ist in der Partikelstruktur `m_particleOdometry` gespeichert, da auch sie Position und Ausrichtung beinhaltet.

Algorithmus 5: void CHohensyburg::moveParticlesByOdometry()

1: **for** $i = 0$ to $i < m_NumberOfParticles$ **do**
2: $validateValue = 1$
3: **if** linker und rechter Torpfosten sichtbar **then**
4: $validateValue$*= Prob(Winkel zwischen den Pfosten ist möglich an der Partikelposition)
5: **end if**{Prüfe weitere mögliche Kombinationen aus Landmarken.}
6: Partikelgewicht $m_Particle\,[i]\,.m_Probability = validateValue$
7: $i + +$
8: **end for**
9: normalisiere $m_Particle\,[i]\,.m_Probability$ so dass $\sum_{i=0}^{n} m_Particle[i].m_Probability = 1$
{Als PDF[10]wurde die Funktion $|180 - x|$ gewählt. Nach Normierung des Ergebnisses auf Das Intervall $[0, 1]$ ergibt sich für den Winkelunterschied $0\,°$ eine Wahrscheinlichkeit von 1 und mit dem Winkelunterschied $180\,°$ ergibt sich 0.
Dieser Vergleich kann für jeweils 2 Winkel mit der Funktion `compareAngles(s1,s2,vs1,vs2)` effizient durchgeführt werden, wobei auch eine alternative PDF benutzt werden kann.
Andere Vergleiche zwischen Sensoren sind möglich, z.B. Distanzen zu den Fußpunkten beim Torwart oder Winkel zu den Eckfahnen}

Algorithmus 6: void CHohensyburg::validateParticlesBySensors()

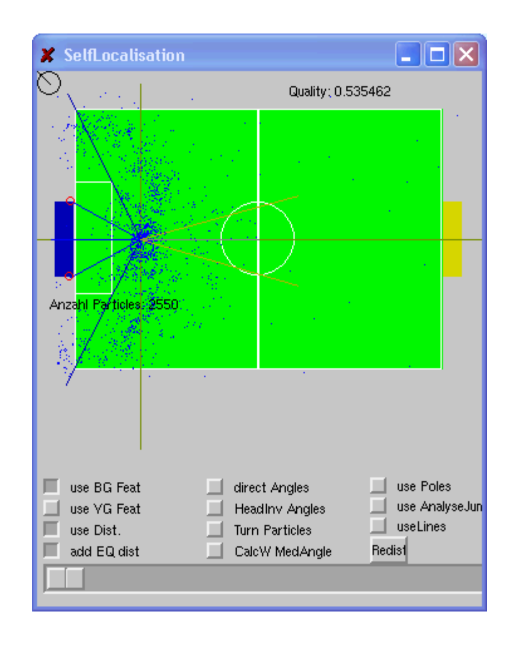


Abbildung 3.12: Beispiel der Verteilung der Partikel nach einem Integrationsschritt und bekannter Entfernung zu den Fußpunkten des blauen Tors.

3.4 Strategie

Bei einem Fußballfeld mit den sich darauf bewegenden Objekten wie Ball, Mit- und Gegenspielern handelt es sich um eine sehr dynamische Umgebung. Hinzu kommt eine hohe Anzahl von Spielsituationen, also Konstellationen von Ball- und Spielerpositionen, in denen ein stets angepasstes Verhalten nötig ist. Ein System, was sich in einer solchen Umgebung zurechtfinden und vernünftige Entscheidungen treffen soll, muss vor allem schnell reagieren können. Daher wurde in diesem Projekt Wert darauf gelegt, eine Programmschleife mit einer möglichst kurzen Zyklus-Zeit zu realisieren. Die Hauptrechenzeit geht dabei zu Lasten der Verarbeitung der Sensorendaten und der darauf aufbauenden Selbstlokalisation. Nur einen kleinen Teil der Rechenzeit verwenden im Vergleich dazu die Routinen der strategischen Entscheidungsfindung und Bahnplanung.

Ein grundsätzliches Problem für die Bahnplanung ist die Tatsache, dass das zugrundeliegende Weltbild aufgrund der Kameradaten niemals eine vollständige Karte der Umgebung widerspiegelt, sondern nur die Objekte innerhalb eines ca. 5m-Radius korrekt darstellt. Dies führt zu der Überlegung, keine vollständige Pfadsuche auf dem gesamten Spielfeld durchzuführen, sondern einen reaktiven Algorithmus zu entwerfen, der zwar aus Unkenntnis der Position weit entfernter Objekte, möglicherweise zunächst eine falsche Entscheidung trifft, kurz- und mittelfristig jedoch schnell und sicher Manövrieren kann und so - auch aufgrund der Dynamik des gesamten Systems - langfristig ans Ziel gelangt.

Man kann die Aufgaben des Strategie-Moduls generell in zwei Teilbereiche zerlegen. Zum einen soll es für eine sichere und robuste Ansteuerung des Roboters zu jedem Zeitpunkt während des Spiels sorgen. Es soll also Ansteuerungsbefehle generieren, die es dem Roboter ermöglichen sicher zwischen Hindernissen hindurch zu manövrieren oder etwa den Ball bei Kurvenfahrten zu führen. Diese Fähigkeiten werden im Folgenden Grundfähigkeiten genannt und bilden die Basis zur Ansteuerung des Roboters. Zum anderen soll der Roboter intelligentes Spielverhalten zeigen, d.h. er soll den verschiedenen Spielsituationen angepasste Manöver fahren und z.B. nicht auf das eigene Tor spielen . Diese Eigenschaften werden im Folgenden als höhere Fähigkeiten bezeichnet.

3.4.1 Grundfähigkeiten

Ein Ziel für das Strategie-Modul ist die Bereitstellung von robusten Grundfähigkeiten. Dazu gehören:

1. zuverlässiges Manövrieren auf dem Feld mit und ohne Ball
2. stabile Ball-Anfahrt
3. Hindernis-Kollisions-Vermeidung (Obstacle Avoidance)
4. schnelles Ball-Dribbling

Diese Fähigkeiten besitzt der Roboter aus jeder Spielsituation heraus, unabhängig von seiner Position, Ausrichtung, Geschwindigkeit und Fahrtrichtung.

Da die Positionen des Balls und der Hindernisse stets relativ zum Roboter vorliegen, funktionieren die oben genannten Grundfähigkeiten prinzipiell auch bei schlechter oder sogar ausgefallener Selbstlokalisation.

Die Verwendung eines omnidirektionalen Fahrwerks verschafft diverse z.T. auch taktische Vorteile (z.B. kürzerer und direkterer Ballanfahrtsweg). Diese sollten bei der Umsetzung der Grundfähigkeiten genutzt werden.

Zur Realisierung der o.a. Aufgaben kommen zwei Ansätze zum Tragen. Zentrales Unterscheidungskriterium ist dabei der Ballbesitz. Besitzt der Roboter den Ball nicht, d.h. liegt dieser nicht direkt vor seiner Ballführungseinheit und der Roboter muss diesen nicht vor sich her führen, kann er mit ganz anderen Mitteln auf dem Spielfeld manövrieren als mit Ball.

Manövrieren ohne Ball

Für das Manövrieren ohne Ball sind die Grundfähigkeiten ausschließlich auf Basis eines reaktiven Vektorfeldes realisiert. Es werden dazu in jedem Regelzyklus sukzessive die verschiedenen, den Pfad beeinflussenden, Kräfte aufsummiert. Am Ende erhält man den resultierenden Fahrtvektor. Im Einzelnen werden aufsummiert:

Basisvektor in Richtung Ziel + Vektoren der Hindernisse + Anfahrtsvektor = Fahrtvektor

Der Basisvektor ist der Richtungsvektor von der aktuellen Roboterposition zum Ziel. Seine Länge ist stets auf 1 normiert.

Obstacle Avoidance Jedes Hindernis ist von einem gedachten, runden abstoßenden Kraftfeld umgeben, dessen Stärke nach außen abnimmt. Gerät der Roboter in den Einflussbereich des Hindernisses, wird seine geplante Fahrtrichtung (Basisvektor) von dem Hindernis beeinflusst (siehe Abb. 3.13). Er wird vom Hindernis weggedrückt bzw. in mehreren Zyklen herum geleitet.

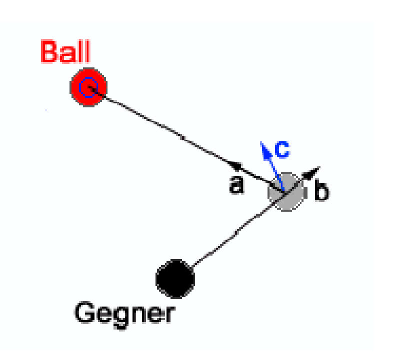


Abbildung 3.13: Einfluss von Hindernissen auf den Fahrtvektor. Beeinflussung des Basisvektors (a) durch den Vektor eines Hindernisses (b) und neuer resultierender Fahrtvektor (c).

Ball-Anfahrt Ähnlich wie beim Hindernis umgibt auch den Ball ein rundes abstoßendes Vektorfeld (im Folgenden Anfahrtsvektorfeld genannt); allerdings mit der Ausnahme, dass dieses Vektorfeld an einer Seite eine Schneise aufweist, durch die der Roboter an den Ball gelangen kann. Man erreicht dadurch, dass der Roboter den Ball von einer bestimmten Seite her anfährt.

Hier nutzt man im übrigen die Eigenschaften des omnidirektionalen Fahrwerks aus, das es dem Roboter erlaubt, permanent zum gegnerischen Tor ausgerichtet zu sein (siehe Abbildung 3.14) und dennoch mit Hilfe des Anfahrtsvektors so um den Ball zu manövrieren, dass er ihn von einer günstigen Seite her anfährt.

Geschwindigkeitskorrektur Völlig unabhängig von der Berechnung des Richtungsvektors des Fahrbefehls wird die entsprechende Geschwindigkeit für den Fahrbefehl ermittelt. Generell versucht der Algorithmus alle Manöver mit maximaler Geschwindigkeit zu fahren. Befindet sich der Roboter jedoch im Einflussbereich von Hindernissen oder in der Nähe des Balls, wird die Geschwindigkeit gedrosselt, um Kollisionen zu vermeiden. Dieser Drosselungsfaktor ist abhängig von der Entfernung

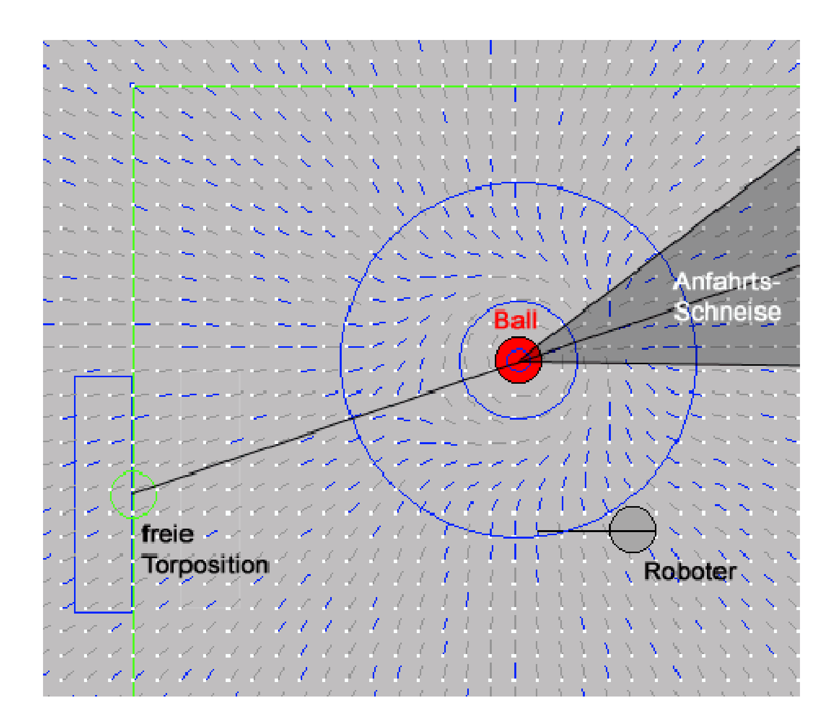


Abbildung 3.14: Anfahrtsvektorfeld des Balls. Der Roboter wird um den Ball herum zur Anfahrtsschneise gelenkt, die in einer Linie mit der freien Position im Tor liegt. Während dieses Manövers behält der Roboter die Ausrichtung zur freien Torposition permanent bei.

zum Hindernis und wird nach außen hin wie beim abstoßenden Vektorfeld ebenfalls schwächer.

Mithilfe der angewendeten reaktiven Vektorfeldmethode ist es nun also möglich, auf dem Spielfeld jedes beliebige Manöver ohne Ball durchzuführen.

Diese Methode ist in der Klasse `CPathFinder` implementiert. Diese Klasse stellt einen Befehlssatz zur Verfügung, über den sämtliche Ansteuerungsparameter des omnidirektionalen Fahrwerks auf einfache Art manipuliert werden können.

Es existieren folgende Befehle:

- `moveToPosXY` Der Roboter fährt zu einer Position auf dem Spielfeld. Dabei hält er Ausrichtung und Anfahrtsvektor gemäß den Eingaben durch `setApproachPoint` und `setOrientation` ein.
- `stop` Der Roboter hält an.
- `setOrientation` Der Roboter richtet sich auf den übergebenen Punkt auf dem Spielfeld aus. Bei allen folgenden Fahrbefehlen behält der Roboter diese Ausrichtung bei. D.h. er gleicht eine falsche Ausrichtung permanent durch Eigenrotation aus.
- `setApproachPoint` Der Anfahrtsvektor wird aus dem übergebenen Bezugspunkt und der Position des Ziels berechnet. Bezugspunkt und Zielpunkt bilden eine Gerade. Auf dieser Geraden - vom Bezugspunkt aus gesehen hinter dem Ziel - liegt der Anfahrtsvektor. Beispiel: Zielpunkt ist der Ball, Bezugspunkt das blaue Tor. Dann fährt der Roboter den Ball stets von der dem blauen Tor abgewandten Seite an.
- `resetApproachPoint` Bei zurückgesetztem Anfahrtsvektor fährt der Roboter den Zielpunkt immer direkt an und zwar von der Seite, von der er sich ihm gerade nähert.

Manövrieren mit Ball

Zum Führen des Balls stehen dem Roboter in der vorliegenden Bauart lediglich zwei flexible, ca. 10 cm-lange Hörnchen an einer Flanke zur Verfügung (siehe auch Kapitel 3.4). Der Ball wird weder festgeklemmt noch irgendwie sonst am Roboter gehalten, d.h. er ist völlig frei beweglich. Hat der Roboter mit Hilfe der oben beschriebenen Ballanfahrtsroutine den Ball einmal zwischen die beiden Führungshörnchen gebracht, muss er ihn nun in eine möglichst günstige Schussposition dribbeln. Die wichtigste Grundregel für dieses Dribbeln des Balls ist, dass jede größere Abnahme der Fahrtgeschwindigkeit den Verlust des Balls bedeutet, da dieser ja kontinuierlich weiterrollt. Der Roboter muss also während er den Ball vor sich her schiebt möglichst dauernd beschleunigen oder zumindest mit konstanter Geschwindigkeit fahren.

Obstacle Avoidance Genau dieses Verhalten wird mit der oben beschriebenen reaktiven Vektorfeldmethode nicht gefördert. Jedes Zunahekommen an ein Hindernis bedeutet eine Drosselung der Geschwindigkeit. Aus diesem Grund wird für den Fall, dass der Roboter den Ball besitzt und führen soll eine Modifikation der reaktiven Vektorfeldmethode vorgenommen. Dazu werden sämtliche Hindernisse im Algorithmus außer Acht gelassen. D.h. es wird lediglich der Basisvektor in Richtung Ziel berechnet, welcher dann direkt als neuer Fahrtvektor benutzt wird. Mit diesem Fahrbefehl würde der Roboter nun sämtliche Hindernisse rammen. Um dies zu vermeiden, wird für die Obstacle Avoidance eine alternative Methode eingesetzt.

Da die Führungshörnchen - und im übrigen auch die Schusseinheit - nur an einer Flanke des Roboters angebracht sind, kann der Roboter den Ball folglich auch nur in diese Richtung dribbeln. Um nun eine Kollision beim Dribbeln zu vermeiden, wird in jedem Zyklus geprüft, ob der Korridor in Richtung Ziel frei ist (siehe Abb. 3.15 links).

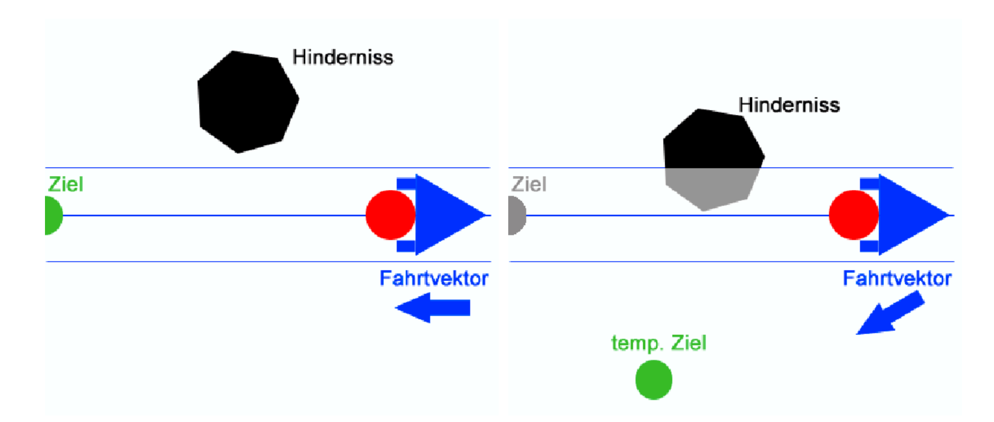


Abbildung 3.15: Obstacle Avoidance. In jedem Zyklus wird überprüft, ob der Korridor in Richtung Ziel frei ist. Ist der Korridor nicht frei wird solange ein verlagertes, temporäres Ziel angefahren, bis er frei ist.

Ist der Korridor durch ein Hindernis blockiert, wird ein temporäres Ziel als Ausweichpunkt seitlich des Hindernisses ermittelt. Dieser neue Zielpunkt wird solange angesteuert, bis das Hindernis umfahren und der Korridor in Richtung des ursprünglichen Ziels wieder frei ist (siehe Abb. 3.15 rechts).

Dribbling Eine weitere Grundfähigkeit ist das schnelle Dribbeln des Balls über das Spielfeld. Mit der oben beschriebenen Ballführungseinheit ist ein Dribbeln des Balls bei Geradeaus-Fahrt relativ problemlos. Schwierig ist hingegen das Halten des Balls bei Kurven-Fahrten. Hierzu wurde ein spezieller Dribbel-Algorithmus entwickelt. Dieser berechnet abhängig von der Winkeldifferenz zwischen

Orientierung und Ziel einen Auslenkungspunkt, der neben dem Ball liegt. Dieser wird bei gleichzeitiger Rotationsbewegung des Roboters angesteuert und führt zu einem Manöver, das die Trägheit und das Rollverhalten des Balls beim Fahren berücksichtigt und verhindert, dass der Ball verloren geht. Bei einer starken Richtungsänderung (große Winkeldifferenz) liegt der Auslenkungspunkt weit neben dem Ball, die Rotationsgeschwindigkeit ist entsprechend hoch. Ist die Winkeldifferenz nur minimal, liegt der Auslenkungspunkt quasi im Ball und die Rotationsgeschwindigkeit ist sehr klein. Dies entspricht dann lediglich kleineren Kurskorrekturen. Das Prinzip des Algorithmus ist in Abb. 3.16 erläutert.

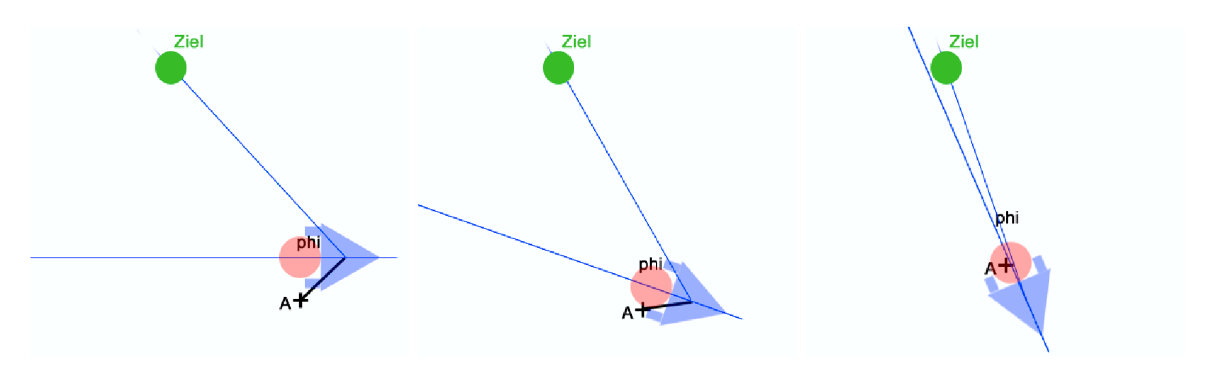


Abbildung 3.16: Dribbling-Algorithmus. Zum Zeitpunkt $t = 0$ (links) ist die Winkeldifferenz φ zwischen Orientierung und Ziel groß. Der Auslenkungspunkt A wandert weit neben den Ball. Gleichzeitig dreht sich der Roboter mit hoher Geschwindigkeit um die eigene Achse (in diesem Bsp. im Uhrzeigersinn). Bei $t = 1$ (Mitte) ist die Winkeldifferenz φ geringer. Der Auslenkungspunkt A liegt näher am Ball. Die Rotationsgeschwindigkeit verringert sich. Die gefahrene Kurve wird weiter. Schließlich zum Zeitpunkt $t = 2$ (rechts) hat der Roboter die neue Ausrichtung erreicht. Die Winkeldifferenz φ ist nahezu 0. Die Kurve ist vollendet und geht in eine nur noch mit leichten Korrekturen versehene Geradeaus-Fahrt über.

Der Vorteil im Vergleich zur Vektorfeldmethode zum Ausweichen von Hindernissen liegt darin, dass die beiden hier verwendeten Routinen zur Obstacle Avoidance und gleichzeitigem Dribbeln es erlauben, eine ständige Vorwärtsbewegung beizubehalten, so dass der Ball gehalten werden kann.

Mit Hilfe der zusätzlichen Methoden ist es nun also auch möglich, den Roboter auf dem Spielfeld zu manövrieren und dabei gleichzeitig den Ball vor sich herzudribbeln.

Zusammenfassung

Man ist nun in der Lage den Roboter in jeder denkbaren Spielsituation beliebig anzusteuern; und zwar grundsätzlich unabhängig von der Geschwindigkeit, Ausrichtung und Position des Roboters. Wählt man dennoch sehr harte Parameter aus, z.B. zwei aufeinanderfolgende völlig entgegengesetzte Zielkoordinaten, die angedribbelt werden sollen, werden diese Eingaben in realisierbare Manöver umgesetzt.

Wann welche der beiden vorgestellten Manövrier-Methoden angewendet wird, entscheidet sich lediglich dadurch, ob der Roboter im Besitz des Balls ist oder nicht. Dieses Kriterium wird aufgrund der Nähe des Balls zum Roboter in einem bestimmten Winkelbereich vor seiner Dribbel-Einheit bestimmt (siehe Abb. 3.17).

Die Grundfähigkeiten müssen nun allerdings noch sinnvoll angewendet werden. Für diese Aufgabe existiert eine weitere Klasse, die die höheren Fähigkeiten des Systems implementiert.

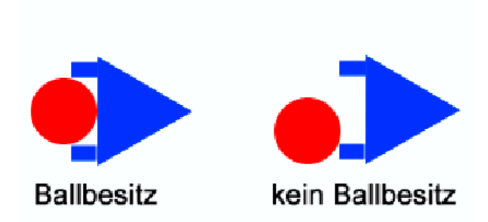


Abbildung 3.17: Unterscheidungskriterium Ballbesitz

3.4.2 Höhere Fähigkeiten

Aufbauend auf den Grundfähigkeiten, die es ermöglichen, dem Roboter in jeder erdenklichen Lage einen gezielten Ansteuerungsbefehl zu geben, ist es nun nötig, solche Ansteuerungsbefehle zu finden, die taktisch und spieltechnisch möglichst sinnvoll sind.

Dabei ist zu berücksichtigen, dass es verschiedene Spielertypen wie Angreifer und Verteidiger gibt, für die unterschiedliche Entscheidungen getroffen werden müssen.

Die für dieses Projekt aufgestellte Mannschaft verfolgt eine eher defensive Taktik. Die Standard-Aufstellung (siehe Abb. 3.18) für ein Spiel besteht aus einem Stürmer, zwei Verteidigern und einem Torwart. Der Angreifer (ATTACK1) agiert auf dem gesamten Feld. Der Rechts-Außen-Verteidiger (DEFEND1) deckt den rechten, der Links-Außen-Verteidiger (DEFEND2) den linken Bereich der eigenen Spielfeldhälfte ab. Beim Ausfall bestimmter Spielertypen werden automatisch abweichende Aufstellungen eingenommen (siehe Abschnitt 3.5). Alle im Folgenden beschriebenen Routinen beziehen sich ausschließlich auf die Feldspieler. Da sich die Entscheidungsfindung dieser Spielertypen von der des Torwarts grundlegend unterscheidet, werden die Algorithmen des Torwarts in Unterabschnitt 3.4.3 gesondert behandelt.

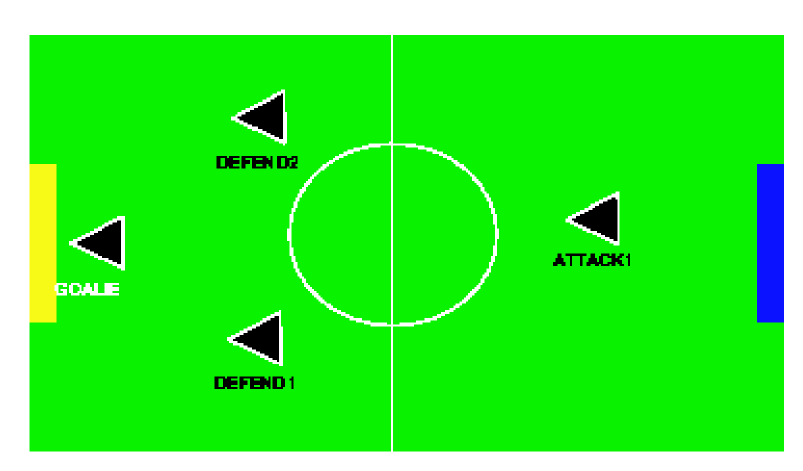


Abbildung 3.18: Standard-Aufstellung des Teams

Grundlage für die Entscheidungsfindung bildet ein binärer Entscheidungsbaum (siehe Abb. 3.19). Zunächst werden für jeden Spielertypen vier Standard-Situationen unterschieden:

1. Anstoß für die gegnerische Mannschaft
2. Anstoß für die eigene Mannschaft
3. Freistoß für die gegnerische Mannschaft

4. Elfmeter für die eigene Mannschaft

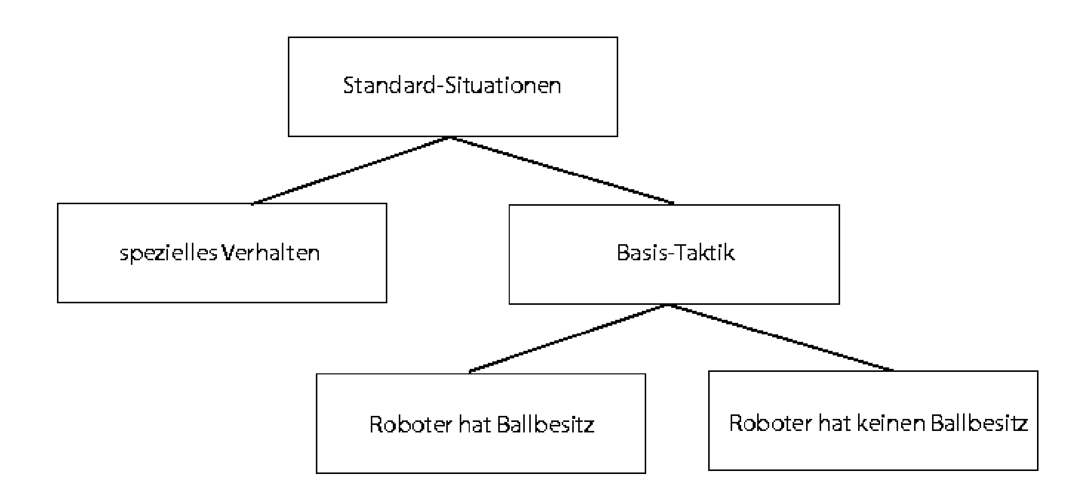


Abbildung 3.19: Hauptäste des binären Entscheidungsbaums der Strategie

Diese vier Situationen verlangen ein spezielles Verhalten von jedem Spielertyp. So führt zum Beispiel der Stürmer beim eigenen Anstoß den ersten Ballkontakt aus, während die Defensiv-Spieler weiter hinten im Feld auf Abwehrstellung gehen. Ähnlich unterschiedliche Verhaltensweisen liegen in den anderen Standardsituationen vor.

Für alle anderen Spielsituationen greift der Algorithmus auf die Manöver der Basis-Taktik zurück. Dieser ebenfalls als binärer Entscheidungsbaum implementierte Teil der Entscheidungsfindung unterscheidet auf oberster Ebene, ob der Roboter im Ballbesitz ist oder nicht. Danach richtet sich sein Verhalten.

Ballbesitz Besitzt der Roboter den Ball, verhält er sich - auch, wenn er eigentlich ein Verteidiger ist - grundsätzlich wie ein Stürmer und versucht, den Ball ins Tor zu bringen. Es werden dabei die oben beschriebenen Routinen für das Manövrieren mit Ball verwendet. Das Spielfeld ist in verschiedene Bereiche eingeteilt (siehe Abb. 3.20). Abhängig davon, in welchem dieser Bereiche der Roboter den Ball ergattert hat, verwendet er unterschiedliche Manöver, um den Ball in die gegnerische Hälfte zu bringen bzw. ein Tor zu schießen. Zum Beispiel fährt er zunächst auf die rechte, gegnerische Torhälfte zu, wenn er den Ball auf der rechten Seite des Bereich 1 oder Bereich 2 ergattert hat, um dann im Bereich 3 eine starke Drehung zur linken Torhälfte zu vollführen, falls der gegnerische Torwart auf der rechten Torhälfte steht. Die Entscheidung, wann solche oder ähnliche Manöver gefahren werden, hängt also davon ab, in welchem Bereich sich der Roboter mit Ball gerade befindet bzw. in welchem Bereich er Ballbesitz erlangt hat.

Kein Ballbesitz Ist der Roboter nicht im Ballbesitz bleiben die Verteidiger auf ihrer entsprechenden Hälfte des Spielfelds (Rechts-Außen, Links-Außen) und verteidigen das Tor. Sie versuchen den Ball nur abzufangen, wenn er in ihren Aktionsbereich gerät. Der Stürmer hingegen versucht in jedem Fall in Ballbesitz zu kommen, egal wo er oder der Ball sich auf dem Spielfeld befinden. Hier finden für alle Spielertypen die oben beschriebenen Routinen für das Manövrieren ohne Ball Anwendung.

3.4.3 Entscheidungsfindung für den Torwart

Die Anforderungen an einen Torwart in der Robocup-Umgebung sind denen eines echten Torwarts sehr ähnlich. Er muss schnell reagieren, Bewegungsbahnen der Bälle gut einschätzen und sicher bzw.

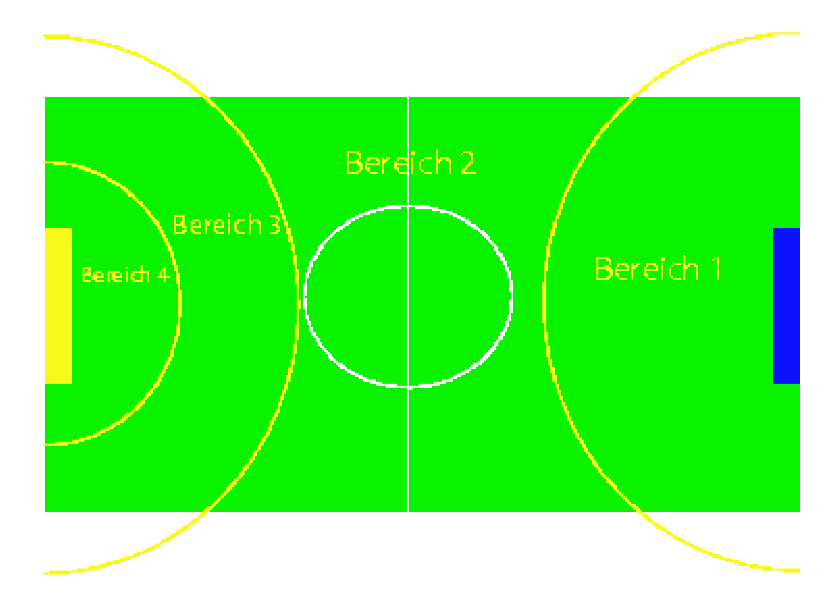


Abbildung 3.20: Einteilung des Spielfelds für Angriffsmanöver beim Spiel auf das gelbe Tor

zuverlässig sein. Von zentraler Bedeutung sind demnach eine robuste Implementierung, eine schnelle präzise Ansteuerung und die Fähigkeit, die Ballbewegung zu erkennen. Letzteres ist besonders wichtig, da jede Entscheidung, die der Torwarts trifft, stark von der Ballbewegung abhängt.

Aktionsraum Vor dem eigenen Tor wird ein Kreisbogen definiert, auf dem sich der Roboter bewegen soll. Der Aktionsraum des Torwarts ist dadurch auf den Bereich, den die Torlinie mit dem Kreisbogen einschließt, beschränkt (vgl. Abbildung 3.21).

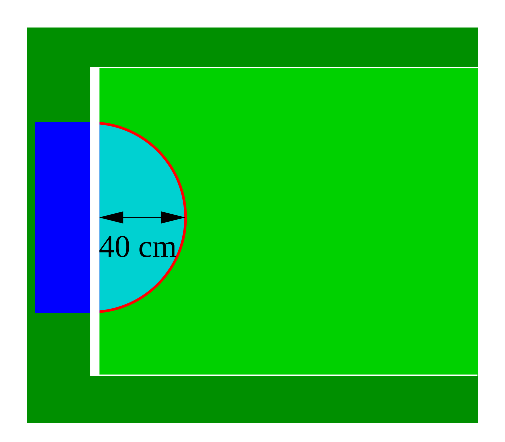


Abbildung 3.21: Aktionsraum des Torwarts. Der Aktionsraum des Torwarts liegt innerhalb des farblich hervorgehobenen Bereichs. Normalerweise bewegt er sich auf dem eingezeichneten Kreisbogen.

In Ausnahmefällen darf dieser Bereich verlassen werden, etwa dann, wenn der Torwart den Ball von hinten anfahren muss, um ihn von der Torlinie zu befördern; oder auch, wenn sich der Ball seit mehreren Sekunden in Strafraumnähe befindet, um den Ball dort zu entfernen. In jedem Fall aber wird der Torwart nie links und rechts über die Pfosten hinaus fahren. Diese Einschränkung trägt wesentlich zur Robustheit bei, da so die Kollision mit den Pfosten und eine damit verbundene Delokalisation vermieden wird. So gab es beispielsweise bei den Weltmeisterschaften 2003 in Padua keinen durch ein solches Problem bedingten Ausfall des Torhüters.

Ballbeobachtung Die relativen Ballkoordinaten bekommt der Torwart in jedem neuen Zyklus aus der Bildverarbeitung; kombiniert mit seiner aktuellen Position auf dem Spielfeld kennt er damit die Ballposition. Bildet man den Vektor von der letzten Ballposition zur aktuellen, so erhält man die Bewegungsrichtung des Balls und die Länge dieses Vektors beschreibt die Ballgeschwindigkeit. Allerdings sind die Daten aus der Bildverarbeitung oft verrauscht und ungenau. Abhilfe schafft ein glättender Filter. Es wurde ein eigener Ballfilter entwickelt (vgl. Abbildung 3.22), der die Linearität des Weges eines rollenden Balls ausnutzt. Ist der Weg, den der Ball nimmt, über mehrere Zyklen hinweg konstant entspricht dies einer hohen Qualität des Ballvektors und der Torwart verlässt sich bei der Entscheidungsfindung auf diese Information. Dies ist bei schnellen Bällen in der Regel der Fall. Ruhende oder langsame Bälle haben oft einen springenden Verlauf, was zu einer dementsprechend niedrigen Qualität führt. Da man sich hier kaum mehr auf die Information des Ballvektors verlassen kann wird der Ball als ruhend betrachtet. Die aktuelle Ballposition, falls vorhanden, wird direkt aus der Bildverarbeitung übernommen.

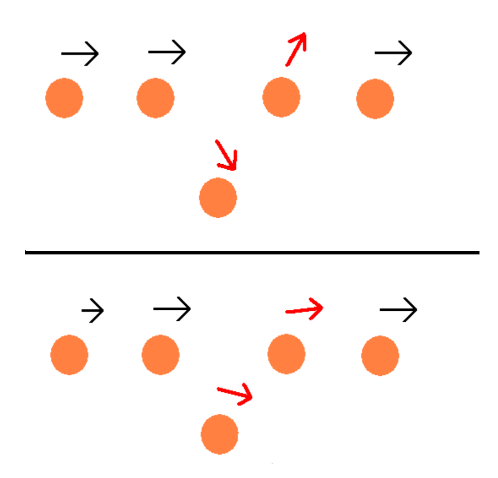


Abbildung 3.22: Wirkungsweise des Ballfilters. Ohne Ballfilter verfälscht eine inkorrekte Information aus der Bildverarbeitung die berechnete Ballbewegung stark (oben). Nimmt man den Ballfilter hinzu, so ist die Berechnung nur leicht verfälscht (unten).

Entscheidungsfindung Abhängig von der Qualität des Ballvektors entscheidet der Torwart nun, ob die Information über die Ballbewegung mit in die Entscheidungsfindung einfließt oder nicht. Falls die Qualität zu niedrig ist wird ein ruhender Ball angenommen und der Torwart positioniert sich so, dass der offene Torraum links und rechts seitlich bzw. hinter ihm minmale Angriffsfläche für einen auf die Tormitte zurollenden Ball bieten würde. Diese Position sei P_1. Bei einem rollenden Ball wird berechnet, an welcher Stelle er die Kreislinie vor dem Tor passieren würde, bezeichnet mit P_2. Abhängig von der Ballgeschwindigkeit und Balldistanz wird nun eine Zielposition berechnet, welche zwischen P_1 und P_2 liegt.

Ansteuerung Da der Torwart in der Lage sein muss, schnell und präzise Positionen anzufahren und sich nicht um eine Hindernisvermeidung bemühen braucht, wird nicht das in Unterabschnitt 3.4.1 beschriebene Vektorfeld zur Ansteuerung benutzt sondern ein eigens dafür entwickeltes Modul . Als Eingabe dient der in der Entscheidungsfindung berechnete Zielpunkt. Abhängig von der Entfernung vom Torwart bis zum Zielpunkt und der maximalen Anfahrts- und Bremsbeschleunigung (es kann schneller gebremst als beschleunigt werden) wird die erlaubte Maximalgeschwindigkeit für den nächsten Ansteuerungsbefehl berechnet. Diese wird an das Beschleunigungsmodul geschickt und von da aus letztendlich an den Roboter.

3.5 Kommunikation

3.5.1 Protokoll und Schnittstellen

Für ein effektives Roboterverhalten ist die Kommunikation untereinander unerlässlich. Die von uns eingesetzte Technologie ist das WLAN nach IEEE 802.11b im lizenzfreien 2.4 GHz Band mit einer maximalen Übertragungsrate von 11 MBit pro Sekunde brutto. Der effektive Durchsatz liegt deutlich darunter, jedoch sind die zu übertragenden Datenmengen für unseren Einsatzzweck erwartungsgemäß im niedrigen KByte-Bereich angesiedelt, so dass diese Technologie unsere Anforderungen erfüllt. Die verlässliche Übertragung der Daten wird mittels verbindungsorientierten TCP[12]-Verbindungen sichergestellt.

Die Kommunikation der Roboter untereinander findet in einer Stern-Topologie statt: die TCP-Verbindungen bestehen ausschließlich zwischen je einem Roboter und dem TribotsControl-Rechner, unmittelbar untereinander bauen die Roboter keine Verbindung auf. Dieses Konzept hat zwei Vorteile gegenüber der $n \leftrightarrow n$ Verbindung:

- eine geringere Anzahl an aufgebauten Verbindungen, 4 gegenüber 10 bei der $n \leftrightarrow n$-Relation
- das $n \leftrightarrow 1$-Konzept beschreibt die zentrale Administrations- und Überwachungsrolle des Tribots-Control-Rechners effektiver und logischer. Dieses wird in den nächsten beiden Unterpunkten näher erläutert.

Die Verbindungen zwischen TribotsControl und den Robotern werden von TribotsControl ausgehend initiiert. TribotsControl sendet eine Verbindungsanforderung an den Roboter, um eine Verbindung aufzubauen. Wenn der Roboter online ist, d.h. das WLAN-Modul funktioniert und das Tribots Programm läuft, akzeptiert er die Verbindung und bestätigt diese durch eine Rückverbindung auf TribotsControl. Zur Implementierung nutzen wir die QSocket und QServerSocket API aus dem QT 3.0.5 Toolkit. Auf der Anwendungsschicht haben wir als Protokoll einen CSV[13]-String definiert, der aus sieben Datenblöcken besteht.

TIMESTAMP	RECEIVER	SENDER	MESSAGECODE	DATAX	DATAY	DATAZ
QTime	QString	QString	int	float	float	float

Tabelle 3.4: Aufbau eines Nachrichten-Strings

`TIMESTAMP` enthält den genauen Zeitstempel zur Verifikation der Datenaktualität, dies ist z.B. nötig bei der Synchronisation der Selbstlokalisationspositionen. `RECEIVER` und `SENDER` enthalten die Absender- bzw. Empfängeradresse. Zur besseren Zuordnung der Roboter werden diese intern im TribotsControl mit einer ID (1-4) angesprochen und vor dem Senden automatisch auf die Empfänger IP abgebildet. Der `MESSAGECODE` beschreibt die Bedeutung der nächsten drei Datenblöcke `DATAX`, `DATAY` und `DATAZ`. In der aktuellen Version gibt es 22 unterschiedliche Befehle und Meldungen, die über die Kommunikationsschnittstelle ausgetauscht werden können. Zur Illustration sei folgender String gegeben:

```
17:34:33, 3, 129.217.57.183, 207, 6, 2, 3
```

`TIMESTAMP` ist selbsterklärend, `RECEIVER` ist die ID 3, der nächste Datenblock gibt die IP des Senders wieder. Der `MESSAGECODE 207` entspricht *“sending CURRENT ROLE, TARGET GOAL and KICKOFF PRESET“*. TribotsControl empfängt diesen String und teilt ihn in sieben nach Kommata getrennte Datenblöcke ein. Der gesamte String bedeutet dekodiert: Roboter 3 spielt als DEFEND3

[12] Transport Control Protocol
[13] comma separated value

(`DATAX=6`) auf das blaue Tor (`DATAY=2`) mit dem KICKOFF PRESET 3 (`DATAZ=3`).

Das aufgestellte Kommunikationsprotokoll bietet Flexibilität, weil unterschiedliche Datentypen (`int, float, QTIME,...`) über einen CSV-String übertragen und auf der Empfängerseite wieder dekodiert werden können. Der Nachrichtencode ist variabel und kann jederzeit leicht angepasst und erweitert werden, um weitere Daten zu übertragen. Diese Flexibilität produziert gegenüber einem minimierten Protokoll Daten-Overhead. Aufgrund der zur Verfügung stehenden ausreichenden Bandbreite der zugrundeliegenden Technologie sind dadurch jedoch keine Nachteile zu erwarten.

Die Verbindung zwischen Roboter und TribotsControl besteht üblicherweise während des gesamten Spiels. Unmittelbar nach Verbindungsaufbau senden die Roboter ihren aktuellen Status (`Role`, `TargetGoal`, `KickOff Preset`) an TribotsControl. Alle 300 ms senden die Roboter zudem ihre aktuelle Selbstlokalisationsposition. Sofern ein Fehler, etwa der Absturz des Kameramoduls, auftritt und bei Änderung der Einstellungen zu Role oder TargetGoal wird sofort eine Nachricht an TribotsControl verschickt. Hier werden alle Daten gebündelt aufgearbeitet und die nötigen Maßnahmen ergriffen. Trotz der Nutzung von TCP-Paketen kann es jedoch u.a. durch erhöhten Verkehr zu andauernden Kollisionen und damit zu Paketverlusten kommen. Das Tribots Software-Framework ist so konstruiert, dass die Roboter auch ohne WLAN- Verbindung reibungslos funktionieren und nicht auf permanente WLAN-Daten angewiesen sind. Das Senden der Nachrichten erfolgt durch die Nutzung der QSocket-API in einem eigenen Thread, so dass Paketverluste und Timeouts nicht zur Störung des gesamten Programmablaufs führen.

3.5.2 Dynamischer Rollenwechsel

Während eines Roboter-Fußballspiels sind Ausfälle der Protagonisten zu erwarten. Zum einen ist die Hardware noch nicht so robust, dass sie jeder physikalischen Belastung standhält, zum anderen kann vom Schiedsrichter ein Roboter wegen Regelverstoßes vom Platz gestellt werden. Jede Veränderung der Anzahl der im Spiel verfügbaren Roboter muss unmittelbar zu einer Überprüfung und Anpassung der Rollenverteilung führen. Jeder Roboter hat zu jedem Zeitpunkt eine bestimmte *Role* zugewiesen, sie definiert für einen Roboter verschiedene Variablen. So wird eine bestimmte Homeposition zugewiesen, genauso wie der Spielfeldbereich, der vom Roboter im Normalzustand angefahren werden darf. Außerdem wird eine Patrolroute festgelegt, die der Roboter abfährt, wenn sein aktuelles Weltbild keinen Ball lokalisieren konnte. Die übliche Rollenverteilung bei vier Robotern ist `GOALIE`, `DEFEND1`, `DEFEND2`, `ATTACK2`.

Sobald während eines Spieles ein Roboter gestoppt wird, sendet er diese Information an TribotsControl. TribotsControl registriert den Ausfall der Spielerrole und errechnet aus den noch verfügbaren Spielern die ideale Aufstellung. So muss z.B. zu jedem Zeitpunkt genau ein aktiver Torwart verfügbar sein. TribotsControl sendet an die Roboter ihre aktualisierten Roles und diese Anweisung wird unmittelbar von den Robotern umgesetzt. Es ist erwünscht, dass ein Roboterausfall möglichst früh erkannt wird um nötige Maßnahmen rechtzeitig zu ergreifen. Aufgrund der physikalischen Belastung der Kamera kann es zu einem Absturz des Kameramoduls kommen, der von der Roboter-Software registriert wird. Das Kommunikationsmodul sendet den Ausfall des Roboters an TribotsControl und ein Wechsel der Rollen wird überprüft und falls nötig an die Roboter kommuniziert.

Analog zu einem Spielerausfall wird auch bei einer Spielereinwechselung evtl. ein Rollenwechsel nötig. Dies ist z.B. der Fall, wenn der dedizierte Torwart wieder zurück ins Spiel kommt und der momentane Torwart wieder seine alte Rolle einnehmen muss. Auch hier nimmt TribotsControl automatisch alle benötigten Wechsel vor.

3.5.3 TribotsControl

TribotsControl ist zum einen die zentrale Kommunikationsschnittstelle der Tribots-Architektur, zum anderen stellt es eine graphische Schnittstelle zur Administration und Überwachung der Roboter bereit. Die Roboter können remote gestartet, gestoppt und zur Homeposition beordert werden. `Role`, `TargetGoal` und `KickOff Preset` können für jeden Roboter eingestellt werden. Die Roboter senden fortlaufend ihre aktuelle Position, die auf einem virtuellen Spielfeld graphisch dargestellt wird. Fehler in der Selbstlokalisation und damit zusammenhängende Verhaltensfehler können so schnell registriert werden.
Die Statusmeldungen und die aktuellen Einstellungen (`Role, TargetGoal, KickOff Preset`) wer-

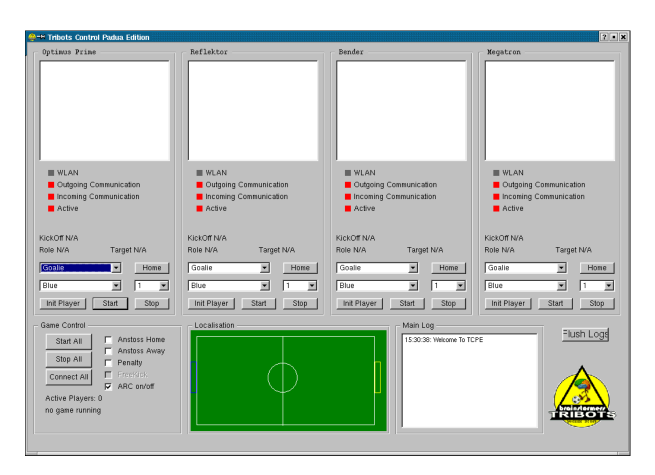


Abbildung 3.23: Tribots Control

den für jeden Roboter einzeln und übersichtlich angezeigt, LEDs zeigen den Verbindungsstatus zu den Robotern an. TribotsControl informiert über berechnete notwendige Rollenwechsel und deren erfolgreiche oder fehlerhafte Durchführung. Vor Spielbeginn und in einer Spielunterbrechung können verschiedene Anstoßvarianten und Spielsituationen gesetzt werden, diese sind `KickOff HomeTeam`, `KickOff AwayTeam` und `Penalty`.

Grundsätzlich gibt es zwei Modi, die das Verhalten von TribotsControl kennzeichnen. Der aktive Modus ist abhängig davon, ob zum aktuellen Zeitpunkt ein Spiel läuft oder nicht. Nach dem Anpfiff des Schiedsrichters werden die Roboter über den Knopf `Start All` gestartet. Jetzt ist TribotsControl solange im Modus *GameRuns*, bis der Knopf `Stop All` gedrückt wird und damit die Roboter und das Spiel angehalten werden. Solange TribotsControl sich im Modus *GameRuns* befindet, wacht die Software ständig über auftretende Roboterausfälle und Robotereinwechselungen, um darauf dynamisch mit adäquaten Rollenwechseln zu reagieren. Ein gestoppter Roboter, sei es aus strategischen Gründen oder aufgrund einer Schiedsrichteranweisung (Regelverstoß), wird sofort von TribotsControl als inaktiver Spieler markiert. Für die übrigen aktiven Spieler wird eine optimale Aufstellung und Rollenverteilung errechnet und per WLAN an die Roboter geschickt. Im Gegensatz hierzu werden die Rollenwechsel nicht vorgenommen, wenn das Spiel nicht läuft bzw. unterbrochen wurde, also *GameRuns = false* ist. Zudem sei angemerkt, dass in diesem Modus die Spieler auch als aktive Roboter im TribotsControl verwaltet werden, wenn sie gestoppt sind. Dies hat den Hintergrund, dass die Roboter schon vor dem unmittelbaren Spielbeginn mit der gewünschten Aufstellung ins Spiel gehen und nicht als Einwechselspieler gehandhabt werden müssen.

3.6 Robotermodul

3.6.1 Überblick

Dieses Modul kapselt alle Hardware-nahen Funktionen und stellt dem Anwender nach außen hin eine einfache Schnittstelle zur Ansteuerung von Motorcontrollern bereit. Bei dem Entwurf dieses Moduls verfolgte man ein modulares Konzept, welches die gesamte Ansteuerung in drei Bereiche (siehe Abbildung 3.24) untergliederte:

- das Kommunikationsmodul ist verantwortlich für die Einhaltung der Echtzeitübertragungen, die Fehlererkennung und stellt rudimentäre Befehle zum Senden und Empfangen bereit
- das Controller-Modul implementiert Controller-spezifische Funktionen und enthält zur Kommunikation notwendige Parameter, z.B. die Schnittstellengeschwindigkeit
- das Wrapper-Modul kapselt das Controller-Modul und stellt dessen Funktionalität nach außen über ein Interface bereit. Dies erlaubt den Austausch des Motorcontrollers, ohne dass sich die Schnittstelle für den Benutzer ändert

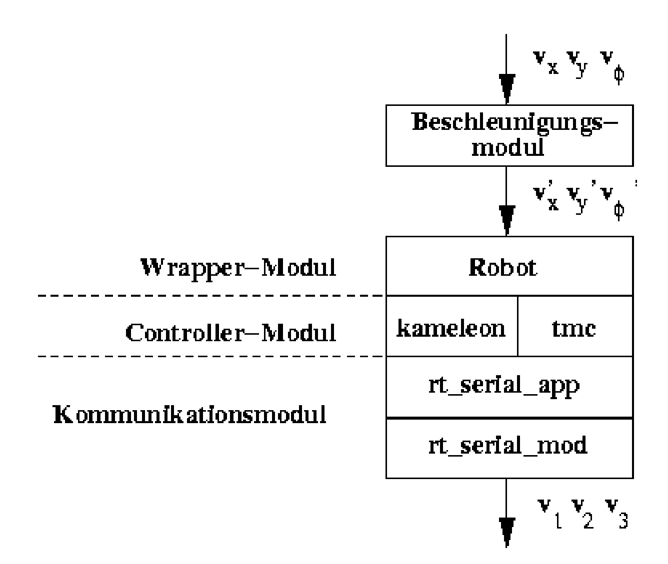


Abbildung 3.24: Komponenten eines Robotermoduls und deren Realisierung. Eingabe sind die Geschwindigkeiten in x- bzw. y-Richtung sowie die Drehgeschwindigkeit v_x-, v_y- und v_φ, welche durch das Beschleunigungsmodul angepasst werden falls nötig. Durch Kinematikgleichungen im Wrappermodul werden diese Geschwindigkeiten auf die einzelnen Räder v_1, v_2 und v_3 umgerechnet und an den Motorcontroller übermittelt.

Mit diesem Ansatz wird eine offene Architektur angestrebt, der Motorcontroller kann ausgetauscht werden, ohne die Schnittstelle nach außen hin ändern zu müssen. Lediglich die Controller-eigenen Befehlssätze erfordern die Implementierung eines neuen Controller-Moduls.

Bevor auf die einzelnen Punkte näher eingegangen wird, folgt eine Einführung in die verwendeten Kommunikationsprotokolle.

Das Kommunikationsprotokoll Der Kommunikation zwischen einem Steuerrechner und dem Motorcontroller muss ein *wohldefiniertes Protokoll* zugrunde liegen, wobei *Befehle klar strukturiert*, gleichzeitig aber so knapp wie möglich gehalten werden sollten. Die *einfache Erweiterbarkeit* des Protokolls um weitere Befehle ist ein Merkmal eines durchdachten Ansatzes. Wie bei jeder anderen

Kommunikation zwischen zwei Endpunkten, gibt es selbstverständlich den Bedarf an *Zuverlässigkeit*, d.h. auf dem Übertragungsweg sollten Nachrichten nicht verloren gehen oder verändert werden. Sollte wider Erwarten trotzdem eine Übertragung modifiziert werden, dann muss ein Mechanismus diesen Fehler erkennen.

Beide Motorcontroller bieten die Möglichkeit der direkten Kommunikation durch das Senden bzw. Empfangen von ASCII-Nachrichten über die RS232-Schnittstelle. Die dabei verwendeten Protokolle weichen voneinander ab und werden in den folgenden Abschnitten separat beschrieben.

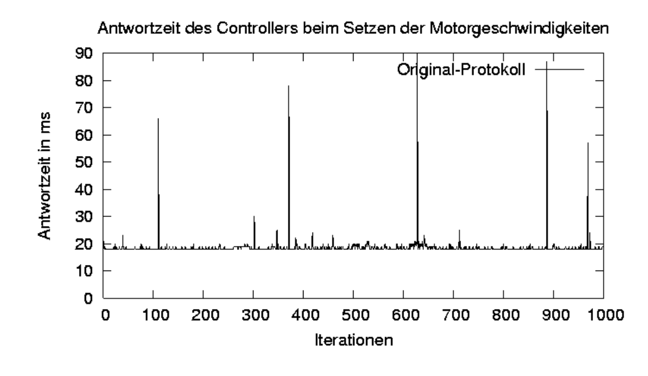
Kameleon Das integrierte Protokoll *SerCom* des Kameleonboards erlaubt die komplette Kontrolle der Boardfunktionalität über die serielle Schnittstelle. Eine genaue Beschreibung der einzelnen Befehle findet sich unter Anhang A in [9]. Durch den Einsatz des REB (siehe Unterabschnitt 2.3.1) erweitert sich der Befehlssatz um die in Anhang B von [10] beschriebenen Möglichkeiten.

Ein Befehl wird ausschließlich vom Steuerrechner abgesetzt und besteht aus einem Großbuchstaben und falls nötig weiteren Zahlen bzw. Literalen, die durch Kommata getrennt werden. Abgeschlossen werden Befehle mit einem Linefeed oder Carriage Return. Die Antwort des Kameleonboards auf diesen Befehl beginnt mit dem gleichen Buchstaben in klein und falls nötig folgen weitere durch Kommata getrennte Zahlen oder Literale.

Dieses Protokoll war für einfache Anwendungen ausreichend, für unser Projekt bedurfte es aber einiger Modifikationen. Der Basisbefehl, der bisher aus einem Zeichen bestand, wurde auf zwei Zeichen erweitert, z.B. gab es für den bisherigen Befehl `D, <Motor>, <Speed>` zum Setzen der Motorgeschwindigkeit eines Motors nun zwei neue Befehle `DA, <Motor>, <Speed>` und `DB,<Speed0>, <Speed1>, <Speed2>, <Speed3>`. Der eine ahmte den Befehl des Original-Protokolls nach und der andere erhielt als Parameter die Soll-Motorgeschwindigkeiten aller Motoren. Dieses Vorgehen wurde möglichst konsistent bei allen Befehlen durchgesetzt.

Das Grundgerüst für das neue Protokoll *LPRSerCom* lieferte John Sweeney[14], Laboratory for Perceptual Robotics, UMASS, Amherst, Massachusets, dem an dieser Stelle besonderer Dank gilt.

Aus eigenen Messungen konnte festgehalten werden, dass die Kommunikation über das selbstentworfene Protokoll Laufzeitvorteile mit sich brachte. Begründen lässt sich dies durch die verminderte Anzahl an Verbindungsaufbauten zwischen Steuerrechner und Motorcontroller. Um beim Beispiel des Setzens der Motorgeschwindigkeiten zu bleiben: mit dem Original-Protokoll benötigte man drei separate Befehle (inkl. jeweils ein Verbindungsaufbau) mit insgesamt sechs Parametern, verglichen mit einem Befehl, der nur vier Parameter übermittelt um alle Motoren anzusteuern.



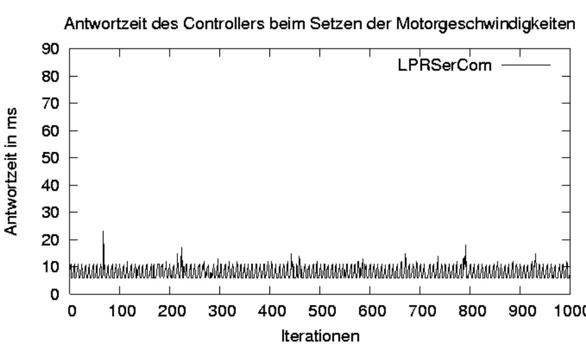


Abbildung 3.25: Zeit zum Setzen der Motorgeschwindigkeiten (Kameleon). In den Graphiken erkennt man die Befehlsdauer in ms für das Setzen aller drei Motorgeschwindigkeiten, 1000 Iterationen wurden jeweils für das Original-Protokoll SerCom (links) und das selbst entworfene Protokoll LPRSerCom (rechts) durchgeführt.

[14] `sweeney@cs.umass.edu`

Im einem Testprogramm wurde jedem Motor 1000-mal eine Geschwindigkeit von 0 m/s zugewiesen. Gemessen wurde die Zeit für das Setzen aller drei Motorgeschwindigkeiten unter Verwendung der verschiedenen Protokolle. Die durchschnittliche Zeit beim Original-Protokoll lag bei ca. 18 ms, Ausreißer lagen im 50 bis 90 ms-Bereich, was für zeitkritische Anwendungen wie unsere zu hoch ist. Im Gegensatz dazu erreicht unser selbstentwickeltes Protokoll einen Durchschnitt von 8 ms mit Ausreißern im Bereich von 15 bis 30 ms.

An den Abbildungen 3.25 sieht man die zeitliche Differenz noch einmal verdeutlicht. Neben dieser Verbesserung wurden noch weitere Befehle in das Protokoll integriert. Unter anderem waren dies ein Notstop-Befehl, der sofort alle Motorgeschwindigkeiten auf 0 m/s gesetzt hat und ein Befehl zum Sperren der Schusseinheit, auf die im Abschnitt 2.4 genauer eingegangen wird.

TMC Das TMC200 ist ebenso wie das Kameleonboard frei programmierbar, jedoch ist der dafür erforderliche Crosscompiler nicht kostenlos verfügbar. Als Konsequenz hieraus wurde das durch das Fraunhofer Institut vorgegebene Protokoll übernommen. Glücklicherweise ist das Protokoll noch in der Entwicklungsstufe, so dass Änderungswünsche von unserer Seite mit integriert worden sind.

Neben der RS-232 Schnittstelle kann die Anbindung an den Steuerrechner ebensogut über den CAN-Bus geschehen, zum jetzigen Zeitpunkt ist dieser Teil aber noch nicht implementiert. Alle wichtigen Parameter wie Reglerkonstanten, Motorgrößen und Arbeitsmodi können über die jeweilige Kommunikationsschnittstelle eingestellt werden.

Die Kommunikation zwischen Steuerrechner und Controller läuft größtenteils analog zum Kameleonboard: der Rechner übernimmt die Rolle eines Masters und das TMC die Slave-Rolle, wobei nur der Master den Nachrichtenaustausch initiieren darf.

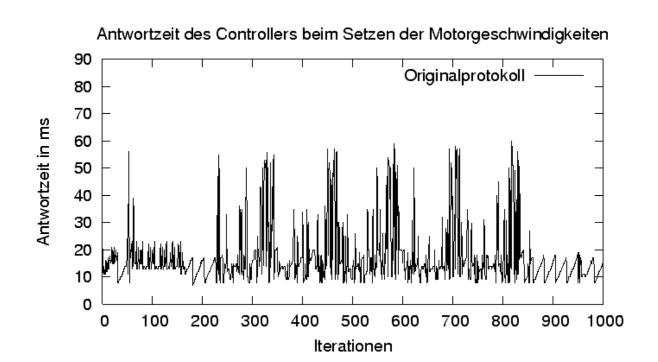


Abbildung 3.26: Zeit zum Setzen der Motorgeschwindigkeiten (TMC). In der Graphik erkennt man die Befehlsdauer in ms für das Setzen aller drei Motorgeschwindigkeiten (1000 Iterationen, Antwortmodus 2). Verglichen mit den ermittelten Werten für den Controller K376SBC (LPRSerCom) stellt man eine recht lange Kommunikationsdauer fest. Begründen lässt sich dies mit dem eingesetzten Antwortmodus[15], da dieser mehr Daten vom Controller zum Host über die Schnittstelle schickt.

Der Befehlssatz des TMC ist klar strukturiert in sog. *Getter* und *Setter*, die entsprechend Parameter auslesen oder setzen. Auf einen Getter wird immer mit den aktuellen Controller-Größen geantwortet und auf Setter wenn er eindeutig identifiziert wurde mit „OK“. Die einzige Ausnahme bildet der Befehl zum Setzen der Motorgeschwindigkeiten dessen Antwort konfiguriert werden kann.

Ein Befehl besteht aus einem Befehlswort, welches ein oder zwei, durch Leerzeichen getrennte Worte enthalten kann. Das erste Wort davon beginnt mit einem „G“ für Getter und „S“ für Setter. Nach dem Befehlswort folgt ein Leerzeichen und anschließend Zahlenwerte. Die Anzahl der Zahlenwerte kann je nach Befehl variieren, wobei die Trennung der Werte erneut durch Leerzeichen erfolgt. Abgeschlossen wird der Befehl mit CR oder LF. Die Antwort des Controllerboards besteht

[15]Der Antwortmodus wurde so gewählt, wie er in einem Spiel wäre.

- für Setter aus dem ASCII-String `OK\n`
- für Getter aus dem Befehlswort ohne das erste Literal gefolgt von den entsprechend dem Befehl geforderten Parametern, jeweils durch Leerzeichen getrennt

So setzt z.B. `SV 0 0 0\n` alle drei Motorgeschwindigkeiten auf 0 m/s und `GMODE \n` liefert den aktuellen Arbeitsmodus zurück.

Kommunikationsmodul Zur Umsetzung der echtzeitfähigen Kommunikation wurde RTLinux, ein Linuxderivat, verwendet. Der bisherige Betriebssystemkern wird dabei als eigenständiger Task in den RT-Kernel eingebunden, welcher „direkt“ auf die Hardware aufsetzt, und erhält die niedrigste Priorität.

Programme, die die Echtzeitfähigkeit nutzen wollen, müssen aus zwei Teilen bestehen: der eine enthält Echtzeit-Aufgaben, z.B. die direkte Kommunikation mit der Hardware und der andere die nicht-zeitkritischen Arbeiten wie das Protokollieren der Daten auf Festplatte.

Die Kommunikation zwischen beiden Teilen wird über eigens dafür erzeugte Character-Devices abgehandelt, sog. RT-FiFos. Für einen bidirektionalen Datenaustausch werden insgesamt vier solcher FiFos benötigt: ein Kommando-FiFo überträgt Befehle aus dem User-Space in den Kernel-Space und der Signal-FiFo überträgt Fehler bzw. Rückmeldungen aus dem Kernel-Space in den User-Space. Für den eigentlichen Datenaustausch zwischen User-Space und Kernel-Space werden zwei weitere FiFos eingesetzt.

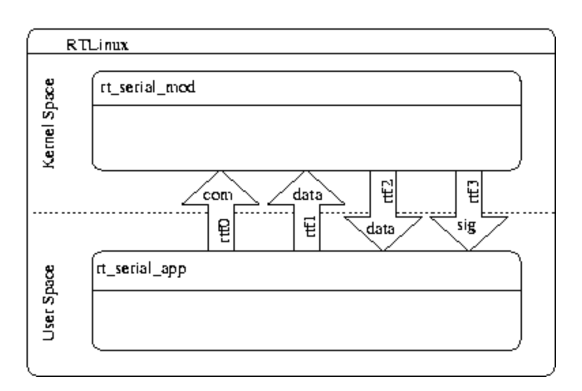


Abbildung 3.27: Kommunikationsschema von RTLinux. Der Echtzeitteil der Anwendung im Kernelspace und der unkritische Teil im Userspace kommunizieren über vier FiFos wie in der Abbildung dargestellt. Zwei der FiFo entsprechen Steuerleitungen und die beiden anderen Datenleitungen.

Realisiert wurde unser Kommunikationsmodul in zwei Teilen, wie bei RTLinux gefordert (vgl. Abbildung 3.27): `rt_serial_mod.c` enthält den echtzeitkritischen Teil, die Kommunikation auf der Schnittstelle und `rt_serial_app.c` übernimmt die restlichen Aufgaben wie Fehlererkennung und Bereitstellung weiterer Funktionen.

Einige der implementierten Funktionen sind

- `rt_talk(int com, char* send, char* reveice, int recv_len, int test)`: Über diesen Aufruf erfolgt die Kommunikation mit der Hardware. Die übergebenen Parameter spiegeln die serielle Schnittstelle, den Sende- und Empfangspuffer, die Länge des Empfangspuffers und die anzuwendende Fehlererkennung wider. Zur Zeit sind mehrere Fehlererkennungsmechanismen integriert, die Controller-spezifisch greifen.
- `block(long delta_t)` und `wait_for_deblock()`: Ein Standard-Linuxsystem hat eine zeitliche Auflösung von 10 ms, was für unsere Zwecke nicht ausreichend war. Unter der Verwendung von

RT-Funktionen wurde eine Auflösung von 1 ms erreicht. Mit Hilfe dieser beiden Funktion wird ein sehr genauer Timer angeboten mit dem ein Benutzerprozess zeitgesteuert werden kann.

`block(...)` sendet über den Kommando-FiFo die Wartezeit `delta_t`, daraufhin wird im Kernel-Space der Timer gestartet und beim Überschreiten des Zeitlimits ein Signal auf dem Signal-FiFo in den User-Space gesendet, auf welches `wait_for_deblock()` wartet.

Controller-Modul Nachdem durch das Kommunikationsmodul ein rudimentäres System zum Nachrichtenaustausch etabliert wurde, ist es nun notwendig eine Controller-spezifische Befehlsimplementierung vorzunehmen. Jede der neuen Funktionen benutzt das Kommunikationsmodul und insbesondere die Funktion `rt_talk(...)` zum Nachrichtenaustausch mit dem Controllerboard.

Jeder der eingesetzten Controller hat seine eigenen Besonderheiten, so dass die Schnittmenge der gemeinsamen Funktionen unterhalb der Controllerboards recht klein ist. Im Wesentlichen beschränkt sie sich auf das Setzen von Geschwindigkeiten, PID-Parametern und dem Auslesen von Daten aus den Inkrementalencodern der Motoren. Einige dieser allgemeinen Funktionen sowie einige spezifische werden nachfolgend näher beschrieben.

- `kam_set_motor_V(kam_data_type* data, int nMotor, int speed)` setzt die Geschwindigkeit des Motors `nMotor` auf den Wert `speed`, die Geschwindigkeitseinheit ist Pulse/10ms.
- `kam_get_pos(kam_data_type* data, int nMot)` liest die Position des 32 bit-Inkrementalencoders von Motor `nMot` in der Einheit Pulse oder A/D-Bits aus.
- `tmc_set_velocity_answermode(tmc_data_type* data, int _mode)` konfiguriert die Antwort auf einen Befehl zum Setzen der Geschwindigkeit, mögliche Werte für `_mode` sind
  - 0: keine Rückantwort
  - 1: aktuelle Geschwindigkeiten
  - 2: + aktuelle Ströme
  - 3: + Nachrichtenzähler, aktuelle Zyklenzeit
  - 4: + Distanz
  - 5 und größer: + aktueller PWM-Output
- `tmc_get_coilTemp(tmc_data_type* data)` liefert für alle Motoren die durch das thermische Modell berechnete Temperatur zurück

Wrapper-Modul Die bisherigen, Controller-spezifischen Befehle sind für eine intuitive und einfache Robotersteuerung nicht ausreichend. Die Funktionen der Controller werden erneut gekapselt, Motivationen hierfür:

Die Geschwindigkeitseinheiten der verschiedenen Controller differieren sehr stark, z.B. Pulse/10 ms des Kameleon und % der Motorleerlaufdrehzahl des TMC200, nach außen jedoch soll für alle Controller eine einheitliche Schnittstelle entstehen, ergo müssen sie alle auf dieselbe Geschwindigkeitseinheit zurückgreifen. Die Entscheidung ist zugunsten von rad/s gefallen, da sie den meisten geläufig ist und eine zweckmäßige Dimension besitzt. Eine analoge Vorgehensweise gibt es für die Versorgungsspannung (normiert auf Volt) und die Motorströme (normiert auf mA).

Über den Controller selbst ist es bereits möglich die Motoren anzusteuern bzw. zu regeln, jedoch ist ihm die Anordnung der Motoren nicht bekannt. Das Wrapper-Modul stellt Funktionen zur einfachen Ansteuerung des omnidirektionalen Antriebs bereit. Zur Umsetzung greifen wir auf die in [5] beschriebenen Kinematikgleichungen zurück.

Bereitgestellte Funktionen sind etwa:

- `robot_set_wheels_V(float vMot1rps,float vMot2rps,float vMot3rps)` wandelt die übergebenen Geschwindigkeiten in die Controller-spezifische Größen um und ruft anschließend den entsprechenden Befehl des Controllers zum Setzen der Geschwindigkeiten auf

- `setXYPVelocity(double _dXPosMP, double _dYPosMP, double _dPhiMP)` setzt die Radgeschwindigkeiten unter Verwendung der Kinematikgleichungen auf die gewünschten Werte in $^{rad}/_{s}$

- `robot_comp_thetap_of_mvel` berechnet die gewünschte Radgeschwindigkeit, um den Roboter in eine Richtung relativ zum Roboterkoordinatensystem fahren zu lassen

3.6.2 Odometrie

Um sich bei der Navigation zurechtzufinden, muss ein Roboter häufig Messungen vornehmen, z.B. wie weit er sich von seinem letzten Standpunkt aus fortbewegt und gedreht hat oder wie seine Radgeschwindigkeiten sind. Als Sensoren für diese Aufgabe wurden Rad-Encoder eingesetzt, von denen es zwei verschiedene Typen gibt:

- absolute Messwertgeber: eine bestimmte Wellenposition wird als Signal in Form eines Codes zurückgeliefert. Dabei benutzt man z.B. Potentiometer, wobei jede Radposition einen eindeutigen Widerstand erzeugt.

- inkrementelle Messwertgeber: entspricht einer Impulsfolge, jedesmal wenn sich das Rad etwas dreht, ändert sich der Zustand von HIGH auf LOW bzw. umgekehrt. Die Anzahl der Impulse pro Zeiteinheit entspricht der Umdrehungsgeschwindigkeit des Rades.

Die Sensoren können zur Messung der Translations- und Rotationsbewegung verwendet werden, theoretisch bestimmt die Integration dieser beiden Werte die aktuelle Roboterposition (Koppelnavigation, dead reckoning). Zum Einsatz kamen bei uns die inkrementellen Messwertgeber sowie das Runge-Kutta-Verfahren zur Integration.

Nach der Integration erfolgt das Verschieben der Verteilung der Aufenthaltswahrscheinlichkeit des Roboters um den Weg x/y und die Rotation φ im Weltmodell. Hierbei werden sowohl durch Ansteuerungen erfolgte Raddrehungen wie auch erzwungene Raddrehungen erfasst und zum Robotermodul zurückgeschickt. Dadurch kann man auch ein Verschieben des Roboters von Hand in der Positionsberechnung berücksichtigen. Leider können durchdrehende Räder nicht erkannt werden, diese können sowohl durch zu große Ansteuerungsänderungen beim Beschleunigen und Abbremsen, als auch durch Kollisionen verursacht werden.

3.6.3 Beschleunigungsmodul

Das Beschleunigungmodul ist die Schnittstelle zwischen Strategie- und Robotermodul. Alle Ansteuerungsbefehle werden hier verarbeitet und anschließend an das Robotermodul weitergeleitet. Ein Ansteuerungbefehl besteht aus einem Translationsvektor (z.B: $(0, 1)$ bedeutet der Roboter soll mit $1\,^{m}/_{s}$ nach vorne fahren) und einem Rotationswert mit welchem man die Eigendrehung steuert (π würde bedeuten: drehe dich in einer Sekunde um $180\,^{\circ}$).

In jedem Takt erhält das Beschleunigungsmodul einen neuen Ansteuerungsbefehl und überprüft, ob er das physikalische Modell des Roboters nicht verletzt. Dadurch ist ein ruckelfreies, weiches Fahren gewährleistet. Folgende Einschränkungen bringt das physikalische Modell mit sich:

- Die Maximalgeschwindigkeit beim geradeaus Fahren beträgt $1.5\,^{m}/_{s}$, beim seitlichen Fahren $1.1\,^{m}/_{s}$

- Es kann mit $2\,^{m}/_{s^2}$ beschleunigt und mit $3\,^{m}/_{s^2}$ gebremst werden

Ein neuer Fahrbefehl kann selten unkorrigiert an das Robotermodul weitergeleitet werden. Richtungsänderungen bei hoher Geschwindigkeit können nicht in einem Taktschritt umgesetzt werden. In diesem Fall wird der alte Fahrbefehl weich in den neuen übergeleitet. Dies geschieht, in dem man den alten Fahrbefehl so lange rotiert und skaliert, bis der neue Fahrbefehl erreicht wird.

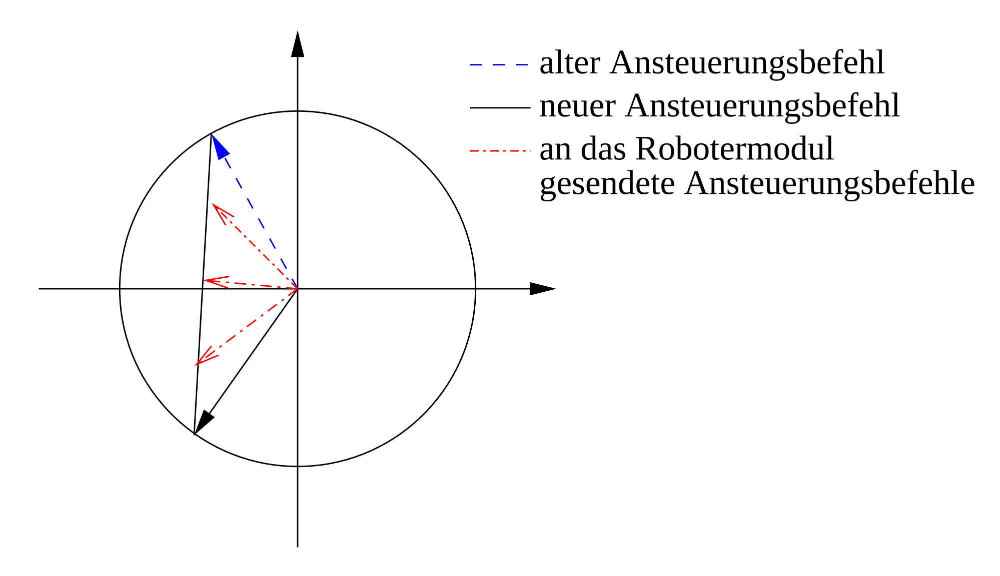


Abbildung 3.28: Modifikation eines Ansteuerungsbefehl durch das Beschleunigungsmodul

3.7 Fremde Software

Innerhalb der Projektgruppe wurden zusätzlich zu der selbst-entwickelten Software weitere Werkzeuge eingesetzt, die das Programmierer-Leben erleichtern sollten.
Neben einer IDE[16] zur Quellcode-Erzeugung wurden z.B. noch ein Dokumentationswerkzeug und eine Quellcode-Verwaltung für sinnvoll gehalten. Die kommenden Abschnitte erläutern welche speziellen Werkzeuge zum Einsatz kamen.

3.7.1 Tools

Zur Verwaltung unseres Projekts setzten wir `cvs` ein. Für eine ausführliche Dokumentation verweisen wir auf `www.cvs.org`.

Die Entwicklungsumgebung, die von uns am meisten genutzt wurde war KDevelop. Einige verwendeten Emacs, aber die meisten Bestandteile wurden mit KDevelop entwickelt. Das Programm und die dazugehörige Dokumentation findet man unter `www.kdevelop.org`.

Die Qt Bibliothek - zu finden unter `www.trolltech.com` - ist eine stabiles und zugleich einfach zu verwendendes Framework für grafische Applikationen jedoch nicht ausschließlich. Es unterstützt Techniken wie TCP/IP, Threads uvm.

Doxygen nutzten wir zur Erstellung von Programm-Dokumentationen. Es stellt verschiedene Ausgabeformate wie HTML, RTF, Postscript, PDF und Unix man-pages zur Verfügung. Doxygen und seine Dokumentation ist unter `www.doxygen.org` zu finden.

Die Kamera wurde über die Bibliothek `cvtk`[7] angesteuert. Dabei nutzt `cvtk` die oben erwähnten Schnittstellen `libdc1394` und `libraw1394` (vgl. Abschnitt 3.2). `cvtk` kann aber nicht nur die Kamera ansteuern, sondern bietet zudem die Möglichkeit farbig markierte Objekte im 2-dimensionalen Raum in Echtzeit zu verfolgen. In der ersten Phase haben wir diese Bibliothek noch genutzt, es wurde aber festgestellt, dass für unsere Zwecke ein eigener Objekterkennungsalgorithmus mehr Performance raushohlen kann. Die Bibliothek `cvtk` ist unter `cvtk.sourceforge.net` erhältlich.

3.7.2 Simulator

Zum Entwickeln von Standard-Manövern wie Torschuss, Dribbling, Ballanfahrt, etc. sind Tests in einer simulierten Umgebung von großem Vorteil. Auch vom allgemeinen Fahrverhalten bekommt man einen schnelleren Eindruck, wenn man einfache Probeläufe nicht am Real-System durchführen muss. Hierfür wurde ein Simulator benötigt, der die für den Robocup typische Umgebung im Rechner modelliert und außerdem eine möglichst einfache Einbindung unserer omnidirektionalen Plattform zuließ. In Frage kamen mehrere Systeme sowie eine komplette Neu-Entwicklung eines solchen Software-Simulators.

Aus Zeitgründen und wegen der einfachen Anpassbarkeit verwendeten wir schließlich den zusammen von der Freiburger und Stuttgarter Universität entwickelten Robocup-Simulator `SimSrv`[17].

Dieser Simulator (siehe Abb. 3.29) modelliert ein Fußballfeld entsprechend den Anforderungen der Robocup-Regeln. Auf diesem Feld können beliebige durch Plugins definierte, Roboter-Modelle herumfahren. Ein physikalisches Modell sorgt für die korrekten kinetischen Reaktionen bei Kontakten mit Gegnern oder dem Ball. Aufgrund der Möglichkeit, mithilfe von frei programmierbaren Plugins eigene Roboter-Modelle in den Simulator zu integrieren, war es innerhalb relativ kurzer Zeit möglich, unser omnidirektionales Fahrwerk und die von uns entworfene Schusseinheit in einem Plugin zu modellieren und so eine Software-Kopie unseres Roboters im Simulator zu testen. Als Eingangssensor wurde das an der Universität Freiburg entwickelte Plugin für die von uns verwendete Kamera Sony

[16] integrated development environment, integrierte Entwicklungsumgebung
[17] `http://kaspar.informatik.uni-freiburg.de/~simsrv`

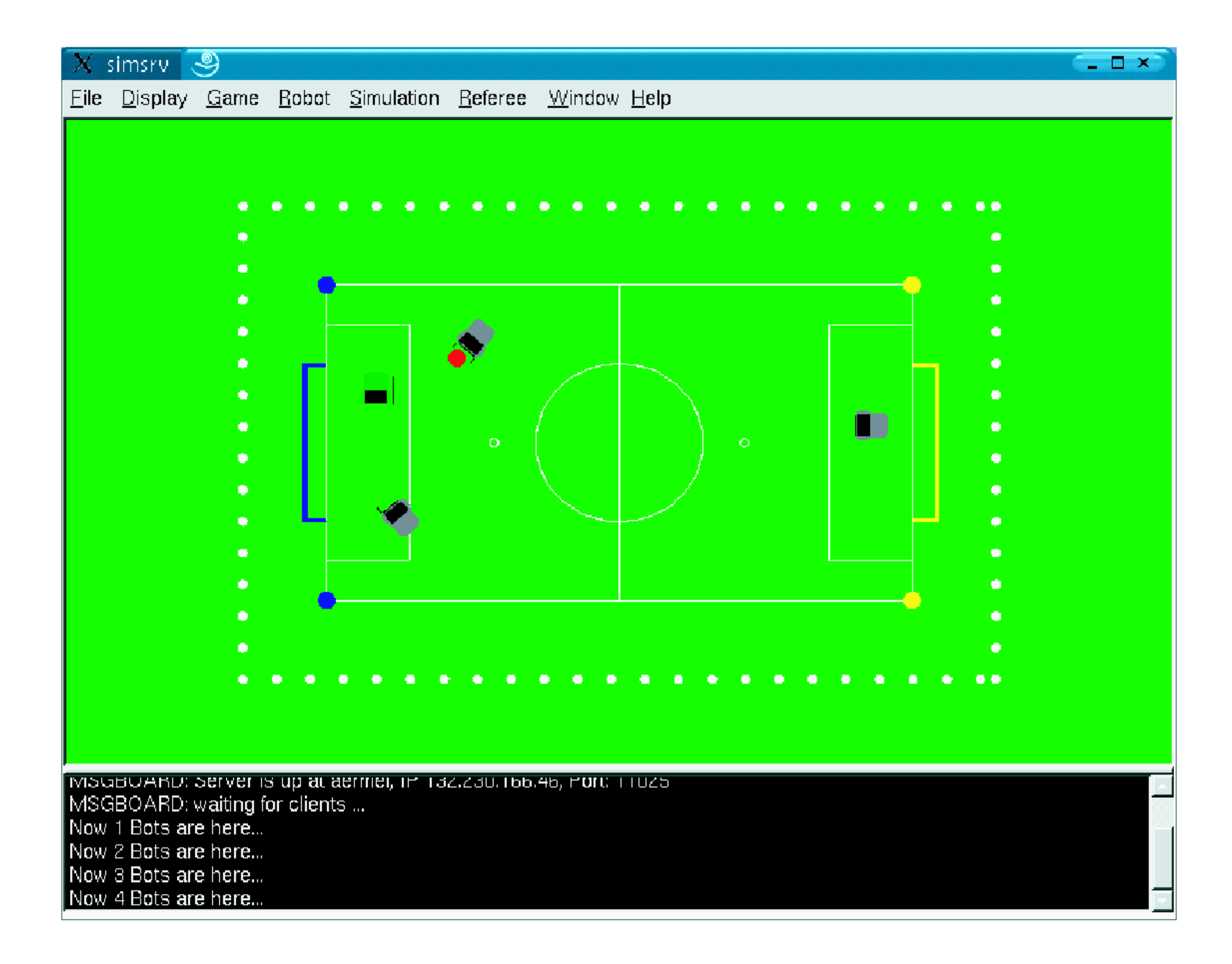


Abbildung 3.29: Oberfläche des Robocup-Simulators `SimSrv`

DFW500[11] verwendet und um das Feature zur Erkennung von Hindernissen erweitert.
Die Anbindung des Freiburger Simulators an unsere Software ist in Abschnitt 3.7.3 beschrieben.

3.7.3 Simulatorplugin

Zur Anbindung der Tribots-Software an den Freiburger Simulator (siehe Unterabschnitt 3.7.2) wurde die Klasse `CDortmundClient` entwickelt.

Anstelle der realen Bilddaten aus der Kamera und den realen Odometriedaten aus dem Controller-Board erhält man bei angeschlossenem Simulator die simulierten Kamera- und Odometriedaten, die denselben Schnittstellenspezifikationen genügen. Ausgangsdaten sind die Steuerbefehle an die Motoren und die Schusseinheit, die ebenfalls bei angeschlossenem Simulator das Roboter-Modell im Rechner anstatt den realen Roboter bewegen.

Es ist grundsätzlich möglich, Ein- und Ausgangsdaten unabhängig voneinander am Simulator zu betreiben. Also z.B. die realen Motoren zusätzlich zu den simulierten Motoren im Rechnermodell anzusteuern und so den echten Roboter parallel fahren zu lassen .

Die Klasse stellt folgende Funktionen bereit:

- `init` verbindet den Simulator mit dem Tribots-Programm

- `setOmniVelocity` setzt die Geschwindigkeit der Motoren im simulierten Roboter-Modell
- `setKicker` aktiviert die Schusseinheit im simulierten Roboter-Modell
- `getOdoData` liefert die Odometriedaten aufgrund der simulierten Ansteuerungsbefehle (interne Berechnung)
- `getObjectArray` liefert ein Datenfeld, in dem alle Features der omnidirektionalen Kamera aufbereitet sind. Dazu gehören: Lage der Eckfahnen, Tore, Ball, Gegner, etc.

Die Verwendung des Simulators eignet sich vor allem für Tests der rudimentären Fahreigenschaften des Roboters. Er wurde daher vor allem für die Entwicklung der oben beschriebenen Grundfähigkeiten eingesetzt. Bei komplexeren Spielsituationen, insbesondere bei Dribbel-Manövern mit Ball, erwies sich der verwendete Simulator als eher ungeeignet, da die Update-Frequenz des physikalischen Modells im Simulator zu gering ist. Die Dauer eines Hauptschleifendurchlaufs der Tribots-Software liegt bei weit unter 100 ms. Der Simulator liefert jedoch nur alle 300 ms neue Umgebungsdaten, so dass feingesteuerte Bewegungen, wie sie z.B. beim Fahren mit Ball notwendig sind, einem zu großen Fehler unterliegen und zu völlig falschen Manövern im Simulator führen.

Kapitel 4

Systemtests

4.1 Bildverarbeitung

Distanzfunktionstests Um die korrekte Funktionsweise der Distanzkorrektur zu überprüfen wurden einige *Benchmarks* durchgeführt. In Abbildung 3.8 kann man die gemessene Entfernung eines Balls in 1 m Entfernung und in 2 m Entfernung erkennen. Nach der Korrektur ist der Abstand in jede Richtung gleich groß. Zusätzlich wurde eine Anfahrt auf einen Ball mit konstanter Geschwindigkeit durchgeführt (siehe Abbildung 4.1). Der jetzt lineare Verlauf der Entfernung ist deutlich zu erkennen. In der direkten Umgebung des Roboters (<40 cm) verschwindet der Ball hinter dem Objektiv. Da der unterste Punkt des Balls zur Berechnung benutzt wurde, tritt hier eine Sättigung der Distanz ein. Für die Schussauslösung wird in diesem Bereich der oberste Punkt des Balls benutzt, die Entfernung dieses Punktes ist im sehr nahen Bereich aussagekräftiger.

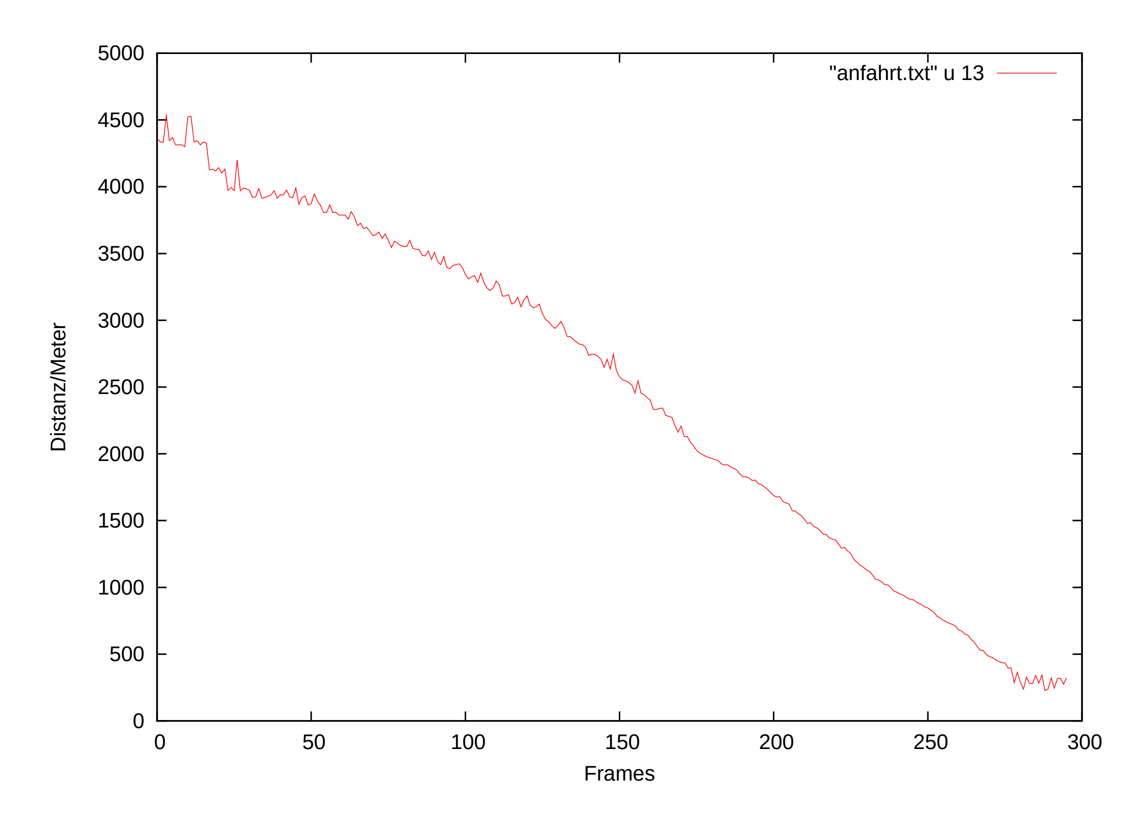


Abbildung 4.1: Anfahrt auf einen Ball mit konstanter Geschwindigkeit

4.2 Selbstlokalisation

Für die Selbstlokalisation wurden mehrere Tests durchgeführt, bei denen der Roboter quer über das Feld mit unterschiedlichen Geschwindigkeiten gefahren wurde. Die Anfangs- und Endpunkte wurden gemessen und die Abweichung von der tatsächlichen Bewegung bestimmt. In den Abbildungen 4.2 und 4.3 sind Ergebnisse solcher Tests dargestellt.

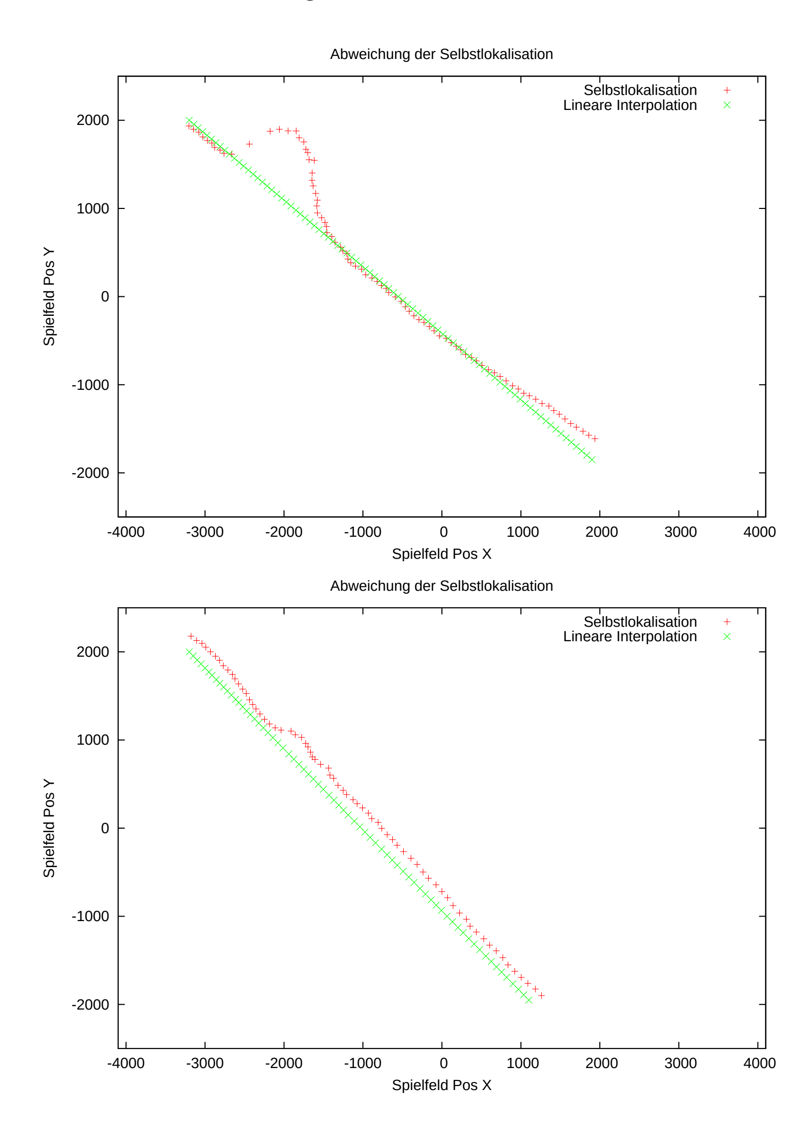


Abbildung 4.2: Fahrt des Roboters quer über das Spielfeld mit 0.5 m/s (oben) und mit 1.5 m/s (unten)

Bei einem anderen Test sollten mehrere definierte Punkte auf dem Feld vom Roboter selbständig angefahren werden. Es wurde eine Umgebung geschaffen, in der aufgezeichnete Bildverarbeitungs- und Odometriedaten benutzt werden, um die Selbstlokalisationsalgorithmen immer wieder mit denselben Daten arbeiten zu lassen, neue Ansätze sind so besser vergleichbar mit bisherigen (*Benchmark*). Aufwendige Tests mit realer Hardware, wobei auftretende Probleme oft schlecht reproduzierbar sind, können mit solchen Testdaten bequem erforscht werden. Im letzten Schritt wurde eine Aufzeichnung der Bilder der omnidirektionalen Kamera ermöglicht, wodurch sowohl Bildverarbeitung als auch Selbstlokalisation *offline* getestet werden können.

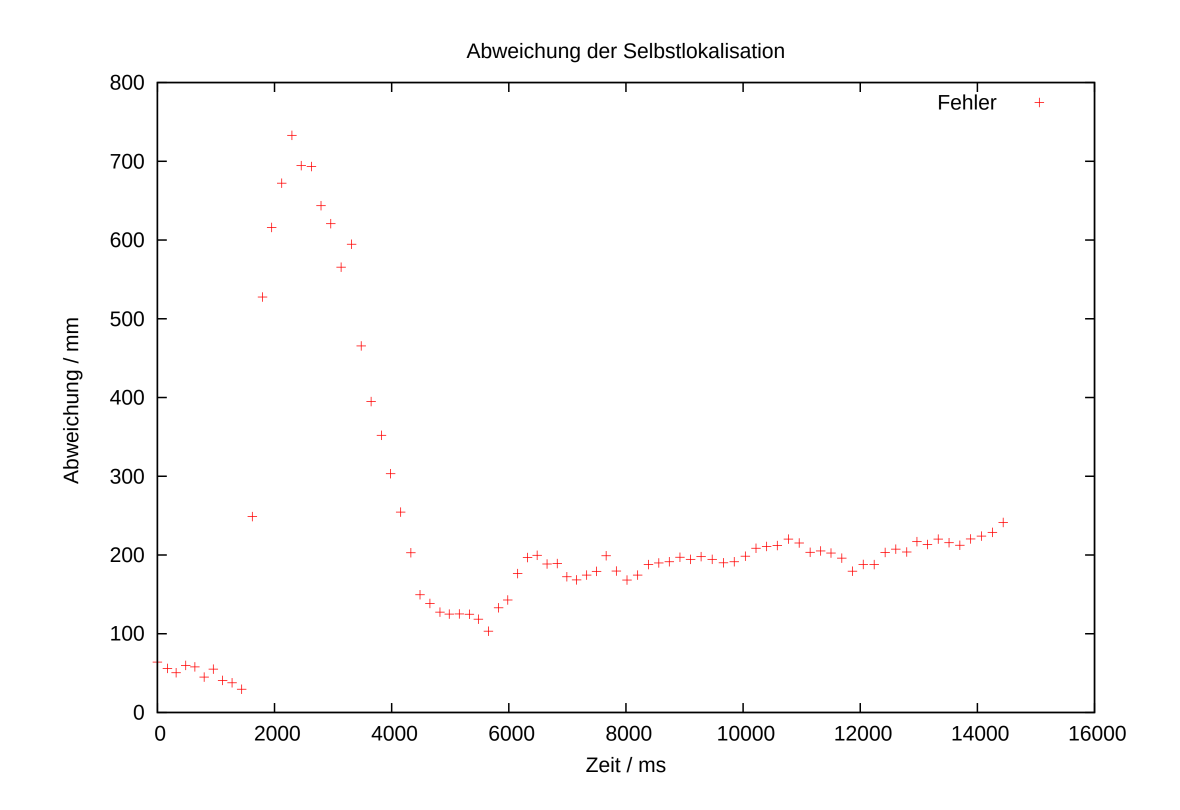


Abbildung 4.3: Abweichungen der Selbstlokalisation von der idealisierten Bewegung mit 0.5 m/s

4.3 Schusseinheit

Zur Weltmeisterschaft sollten alle Roboter mit einer neuen Schusseinheit ausgestattet werden, Voraussetzung hierbei war die Nutzung des Pneumatikzylinders und des Luftdruckbehälters (vgl. Abschnitt 2.4). Getestet wurden verschiedene Aufbauten sowie Komponenten der Schusseinheit. Ziel dabei war, es die mittlere Geschwindigkeit des Balls zu maximieren. Folgende Kombinationen wurden getestet:

1. Schusseinheit 1: Pneumatikzylinder mit Feder

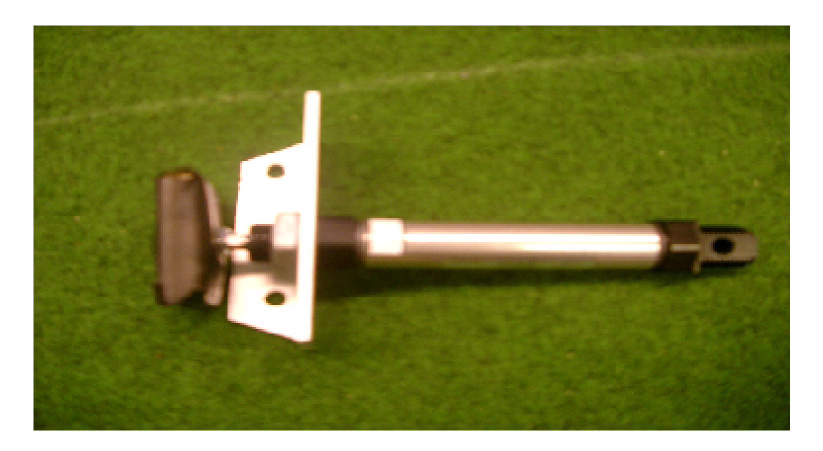


Abbildung 4.4: Schusseinheit bestehend aus einem Pneumatikzylinder mit Feder

2. Schusseinheit 2: Pneumatikzylinder ohne Feder
3. Schusseinheit 3: Pneumatikzylinder mit Feder an Aluminiumhebel



Abbildung 4.5: Schusseinheit bestehend aus einem Pneumatikzylinder ohne Feder

4. Schusseinheit 4: Pneumatikzylinder ohne Feder an Aluminiumhebel

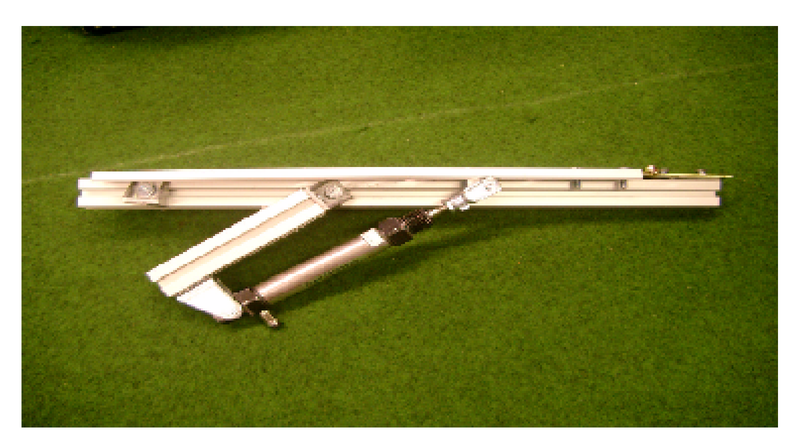


Abbildung 4.6: Schusseinheit bestehend aus einem Pneumatikzylinder (mit / ohne) Feder an einem Aluminiumhebel

- Testaufbau und Durchführung: Der Ball wird direkt an der Schusseinheit anliegend durch Auslösen der Schusseinheit beschleunigt. Gemessen wird die Zeit, die der Ball benötigt, um eine Distanz von 5 m zurückzulegen. Daraus wird die durchschnittliche Geschwindigkeit des Balls berechnet. Jeder Aufbau wurde fünfmal getestet und aus diesen Werten der Mittelwert berechnet.

- Testergebnisse:

	Zeit (s)	Geschwindigkeit (m/s)
SE 1	4,25	1,17
SE 2	3,15	1,58
SE 3	1,5	3,33
SE 4	1,95	2,56

Tabelle 4.1: Verschiedene Schusseinheiten mit Zeit und durchschnittlicher Geschwindigkeit auf 5 m

Bei diesen Ergebnissen fällt auf, dass zwar Schusseinheit 2 den Ball stärker beschleunigt als Schusseinheit 1, da der Pneumatikzylinder ohne Feder einen größeren Hub hat, dieser Vorteil je-

doch durch die Verwendung eines Hebels aufgehoben wird. Durch den Hebel wird die Zeit in der der Ball beschleunigt wird erhöht, so dass hier nicht mehr der Hub des Zylinders entscheidend ist, sondern die Geschwindigkeit der Auslösung des Pneumatikzylinders. Aus diesem Grund haben wir Schusseinheit 3 verwendet, die den Ball auf die höchste Durchschnittsgeschwindigkeit beschleunigt.

4.4 Wettbewerbe

4.4.1 German Open 2003

Im Winter entschloss sich die Projektgruppe zur Teilnahme an den German Open in Paderborn[1]. Diese fanden vom 11. bis 13. April 2003 statt. Die Entscheidung stellte den bis dahin aufgestellten Zeitplan auf den Kopf: bis zu diesem Termin musste eine komplett funktionierende Robotermannschaft fertig sein.

Im Vorfeld wurde daher ein Projektplan erstellt, in dem mehrere Deadlines angegeben waren, zu jeder musste ein neues Merkmal implementiert oder eine neue Ausbaustufe fertiggestellt sein. Die Erfüllung der Kriterien wurde anhand von Benchmarks getestet, die den Fortschritt gut dokumentieren.

Die Veranstaltung selbst war ein Systemtest unter realen Bedingungen. Bis dahin spielten die Roboter ausschließlich alleine oder gegen statische Gegner (Mülltonnen), das völlig neue Wettbewerbsszenario war ein erster echter Test für sämtliche Hard- und Softwarekomponenten. Außerdem musste natürlich die völlig andere Umgebung berücksichtigt werden, besonders die Kalibrierung der Roboter wuchs zu einer völlig neuen Herausforderung. Ein Augenmerk lag auf der Strategie, die sich zum ersten Mal gegen völlig andere Roboter auszeichnen konnte.

Schon zu Turnierbeginn bekamen die Teammitglieder durch die Anwesenheit anderer Teams viele Verbesserungsideen. Die ersten beiden Teams gegen die angetreten wurde, hatten eine schon etwas längere oder genau so lange Entwicklungszeit hinter sich wie unser Team, trotzdem hatten diese mehr Probleme. Es ging gegen das Team der Uni Tübingen und gegen Mostly Harmless aus Graz. Die Roboter dieser Teams waren kaum in der Lage sich zu lokalisieren und bewegten sich nur sporadisch. Die eigenen Roboter jedoch bewegten sich, registrierten ihre Position und die des Balls, so dass die Spiele mit 3:0 und 4:0 recht klar gewonnen wurden.

Im weiteren Verlauf zeigten jedoch einige Gegner die bis dahin noch vorhandenen Schwächen auf, so wie das Team vom Fraunhofer Institut AIS, die wesentlich schneller und agiler waren sowie eine dynamische Rollenverteilung hatten. Das dritte Spiel ging mit 10:0 an sie.

Leider erwiesen sich die eingesetzten Steuerboards vom Typ Kameleon 376SBC als nicht besonders zuverlässig, daher mussten die Roboter öfter als erwartet vom Feld genommen werden um sie neu zu initialisieren. Zudem fielen in diesem Spiel strategische Fehler auf, die am gleichen Abend noch verbessert wurden, z.B. wurde der Torwart sowie die Hinderniseinstellung stark verbessert. Die Strategie wurde insofern überarbeitet, dass die Verteidiger aggressiver zu Werke gingen. Besonders wichtig war, dass ein schwerwiegender Fehler im gesamten Ablauf der Software gefunden wurde, der das Programm stark verlangsamt hatte. Außerdem wurde offensichtlich, dass beim Bau der Roboter die Wirkung der Schusseinheit unterschätzt wurde.

Am zweiten Tag zahlten sich die Änderungen bereits aus, die Leistung verbesserte sich erheblich. Dem bis dahin führenden Team der eigenen Gruppe, IUT Persia aus dem Iran, konnte ein 1:1 abgerungen werden. Dieses Team war auch erst kurz dabei und verfolgte einen komplett anderen Ansatz. Man hatte einen einzigen funktionierenden Roboter, der jedoch sehr schnell war, den Ball gut führen konnte, und zudem noch eine leistungsfähige Schusseinheit besaß, die er in alle Richtungen abfeuern konnte. Man konnte diesem Ein-Mann Sturm mit einer geordneten Abwehr und einem wesentlich stärkeren Torwart entgegentreten. Die Anfälligkeit der Hardware und die Stürmerleistungen

[1] `http://borneo.ais.fraunhofer.de/GO/2003/`



Abbildung 4.7: Vorrundenspiel der German Open gegen Clockwork Orange

verhinderten jedoch, dass dieses Spiel gewonnen werden konnte.

Im nächsten Spiel gegen das etablierte Team Clockwork Orange aus Holland zeigte sich, dass das eigene Team trotz wesentlich kürzerer Entwicklungszeit schon einen großen Schritt getan hatte. Die holländische Mannschaft wurde mit 2:0 besiegt, dabei erwiesen sich besonders Abwehr und Torwart als besser als die der Veteranen, deren Roboter leicht behäbig wirkten. Dies lag aber auch an dem wesentlich älteren Konzept mit Differential-Fahrwerk und einem recht schweren Aufbau. Für das attraktive Spiel mit einer guten Strategie erntete das eigene Team von allen Seiten Lob und Anerkennung, besonders im Hinblick auf die recht kurze Entwicklungszeit.

Trotz dieser Verbesserungen schied das Team in einem äußerst knappen Spiel mit 0:1 unglücklich im Viertelfinale gegen die Sparrows aus Ulm aus[2]. Die Gründe waren offensichtlich, die Hardware verhinderte erneut, dass die Roboter lange einsatzbereit auf dem Feld waren. Auch das Fehlen einer effektiven Schusseinheit wurde schmerzlich vermisst. Weiterhin wurde klar, dass eine dynamische Rollenverteilung das Ausscheiden verhindert hätte, denn ein ausgefallener Stürmer hätte so von einem Verteidiger ersetzt werden können.

Dennoch konnte das Turnier erhobenen Hauptes verlassen werden, einige etablierte Gegner wurden geschlagen und anderen wurden gute Spiele geliefert.

4.4.2 Robocup WM 2003

Nachdem kurz nach den German Open die Entscheidung gefällt worden war zur WM in Padua[3] zu fahren, wurden die Schwachpunkte unseres Abschneidens aus Paderborn zusammengestellt und begonnen, diese in der kurzen Zeit bis zum Beginn des Wettbewerbs möglichst zu beseitigen.

Besonders die instabile Hardware, die Geschwindigkeit der Roboter und die Strategie u.a. im Hinblick auf den Torschuss mussten verbessert werden. Der Torwart sollte komplett überarbeitet werden, zudem bekamen alle Roboter eine leistungsstarke Schusseinheit und die dynamische Rollenverteilung wurde implementiert.

Da im Vorfeld der WM keine Zeit für aufwendige Benchmarks blieb, wurden fast alle neuen Features entweder in selbst erstellten Versuchsaufbauten oder Testspielen gegen die Roboter selbst

[2] Eine detaillierte Auflistung der Ergebnisse im Vergleich zu den anderen Teams: `http://ls1-www.cs.uni-dortmund.de/~merke/robocup/results/ms_go2003_bscolor.html`

[3] `http://www.robocup2003.org`

oder statische Gegner getestet.

Die Weltmeisterschaften fanden vom 5. bis 9. Juli 2003 statt. Wieder eine neue Umgebung, auf die man sich und die Roboter einstellen musste. Während der ersten Spiele wurde klar, dass es sich ausgezahlt hatte auf verschiedene Verbesserungen besonders zu achten.

Im ersten Spiel ging es gegen die bereits bekannten Mostly Harmless aus Graz. Diese hatten sich seit den German Open ebenfalls weiterentwickelt, hatten allerdings immer noch Probleme mit der Lokalisation und setzten daher auf eine sehr defensive Taktik. Der neu implementierte Stürmer konnte dieses Bollwerk jedoch vier mal überwinden, so dass das Spiel am Ende 4:0 ausging.

Der nächste Gegner war genau wie wir ein Newcomer: Die FU Fighters aus Berlin. Das Team an sich hatte schon einige Erfahrung in der Robocup-Umgebung gemacht, die mitgebrachten Roboter waren jedoch fast neu. Diese waren nach einem völlig anderen Prinzip konzipiert als andere, sie waren wesentlich kleiner als alle anderen, hatten aber trotzdem ein omnidirektionales Fahrwerk sowie Kamerasystem. Durch ihre geringe Größe waren sie wendig und sehr schnell. Das Spiel ging unentschieden 1:1 aus und sollte der einzige Punktverlust in der Vorrunde bleiben, denn in den restlichen Spielen wurden die Gegner größtenteils dominiert.

Besonders die Hardware zeigte sich auffallend stabil, das Wechseln des Steuerboards zum TMC200 war die richtige Entscheidung gewesen. Auch die Änderungen an der Software bestätigten sich alle als erfolgreich: die dynamische Rollenverteilung funktionierte auf Anhieb ebenso wie die neuen Strategien, u.a. in Ballführungs- und Torschußsituationen.

Im weiteren Verlauf der Vorrunde wurden weitere Gegner aus Paderborn besiegt, denen man damals noch unterlegen war. So wurden die Sparrows aus Ulm, die in Paderborn noch das Aus besiegelten, mit 6:1 geschlagen. Diese hatten massive Hardwareprobleme, so dass das Ergebnis deutlich ausfiel, da die eigenen Roboter erneut durch Stabilität glänzten.

Auch das folgende Spiel gegen den Angstgegner, das Team vom Fraunhofer AIS, ging 2:0 aus. Dieses Spiel zeigte besonders, wie steil unsere Entwicklungskurve wirklich gewesen war, da in Paderborn noch eine hohe Niederlage hingenommen werden musste.

Im abschließenden Spiel gegen Argus wurde mit 5:0 erneut ein deutlicher Sieg eingefahren, dies war aufgrund der Passivität der Gegner allerdings keine allzu große Überraschung. Diese hatten offensichtlich größere Probleme mit der Lokalisation, die sich bei den eigenen Robotern im ganzen Turnierverlauf als ein großes Plus herausstellte.

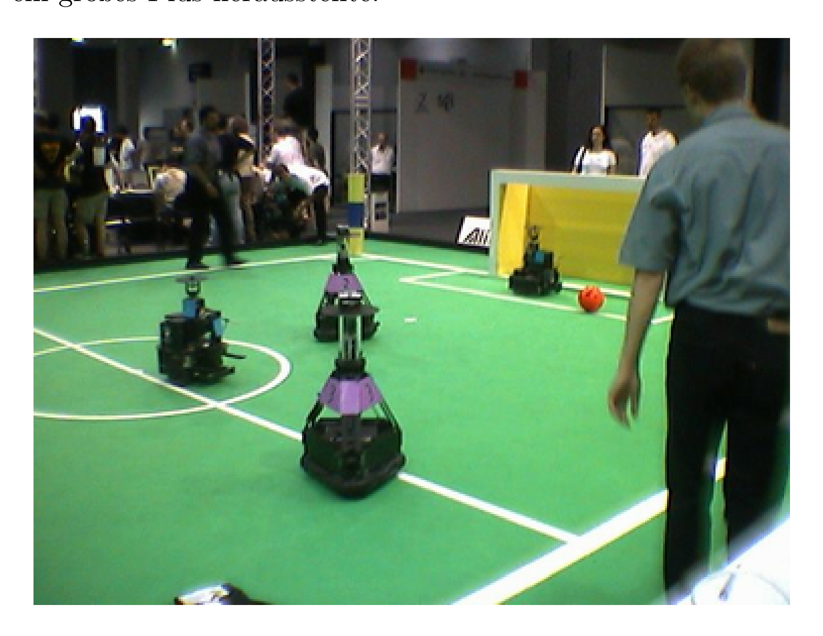


Abbildung 4.8: WM Padua: Zwischenrundenspiel gegen Uni Tübingen

Natürlich fielen nicht nur positive Dinge auf, im Spiel gegen noch stärkere Gegner traten auch

Schwächen zutage. In der Zwischenrunde musste das Team zunächst gegen Eigen, den amtierenden Weltmeister aus Japan antreten. Im Vergleich zu diesem wie auch zu anderen absoluten Weltklasseteams war man immer noch zu langsam und die Ballführung müsste noch verbessert werden. Mit ihren trickreichen Stürmern, die trotz eines Differential-Antriebs schnell und sehr ballgewandt waren, konnten die Gegner die eigenen Bemühungen zunichte machen. Obwohl man zwei mal in Führung ging, ließen die Japaner nicht locker und gewannen am Ende knapp mit 3:4.

Im folgenden Spiel gegen Minho aus Portugal musste daher ein Sieg her, um die Chancen auf das Erreichen des Viertelfinales zu wahren. Leider trat in diesem Spiel eine weitere Schwäche zutage, die sich ansatzweise auch schon gegen Mostly Harmless und Argus angedeutet hatte: Massive Gegner bereiteten den eigenen Robotern große Probleme. Minho spielte im Gegensatz zu den beiden anderen Teams jedoch weitaus cleverer, in dem sie sich besonders und im Grunde ausschließlich in der Defensive auszeichneten. Man fand kein Mittel gegen dieses massive Bollwerk, und dann konnten sie sich mit ihrer überaus starken Schusseinheit einige Male befreien. Zwei von diesen Schüssen landeten unhaltbar im Tor, so dass das Spiel unglücklich mit 0:2 verloren ging.

Damit schied das Team trotz einiger knappen Spiele und einem abschließendem 5:0 Sieg gegen Tübingen im Achtelfinale aus[4]. Nach nur einem Jahr zu den besten 16 Teams der Welt zu gehören und im Achtelfinale der Weltmeisterschaft sehr unglücklich auszuscheiden, war dennoch eine tolle Leistung.

[4]Die Ergebnisse der WM im Vergleich zu den anderen Teams: `http://ls1-www.cs.uni-dortmund.de/~merke/robocup/results/ms_wm2003_bscolor.html`

Kapitel 5

Fazit

5.1 Resumée

Zu Beginn der Projektarbeit, d.h. nach der Orientierungsphase, stand die Gruppe vor der Wahl entweder Lernansätze auf einen Roboter zu bringen – hierbei wäre dann noch zu spezifizieren gewesen, was genau hätte gelernt werden sollen – oder aber ein vollständiges Team von autonomen, Fußball spielenden Robotern zu erstellen, um am Wettbewerb German Open teilzunehmen. Nach der Entscheidung für die German Open wurde mit den Erfahrungen aus der 1. Phase und einer Seminarphase das Hauptaugenmerk darauf gerichtet, ein *ausfallsicheres, schnelles* Team aufzubauen.

Im Vordergrund standen zuverlässige Farbsegmentierung (Objekterkennung), Objekttracking und Monte-Carlo-Selbstlokalisation. Hinzu kam eine reaktive Strategie, die sich aus einfachen Bewegungsfähigkeiten zusammensetzte, um schnell gute Resultate bei der Ansteuerung des Roboters auf dem Spielfeld zu erzielen. Die Erfolge, die das Team bei den German Open verzeichnen konnte, zeigten, dass der Fokus grundsätzlich richtig gelegt wurde.

Mit Ausnahme von einigen Hardwareausfällen erfüllten die Roboter ihre zugewiesenen Aufgaben. Hervorzuheben ist dabei, dass bedingt durch die zuverlässige Objekterkennung fast 100% der Zeit alle relevanten Objekte wie Ball, Tore und Hindernisse erkannt wurden. Somit konnte der Stürmer immer „am Ball bleiben". Dass dies nicht selbstverständlich ist, zeigt sich daran, dass bei anderen Teams zu beobachten war, dass sich die Bilderkennung sehr leicht durch Schuhe, Füße oder Hosen von Zuschauern am Spielfeldrand, die für einen Ball gehalten wurden, verwirren ließ. In den meisten Fällen kommen solche Verwechslungen von einer unsorgfältig eingestellten Farbsegmentierung. Durch die einfache Einstellung der Segmentierungsparameter konnte die Bildverarbeitung auch während einer Auszeit eines Roboters sehr schnell nachgeregelt werden, um akute Probleme zu beseitigen.

Die Roboter konnten sich bei den German Open gut mit einer Genauigkeit von ca. 30 cm lokalisieren. Für alle Roboter ist es wichtig eine möglichst robuste Selbstlokalisation zu besitzen, um die zugewiesenen Rollen zu erfüllen. So war es für unsere Roboter möglich die Startposition selbständig einzunehmen, wodurch man eine korrekte Kalibrierung überprüfen konnte. Alle gegnerischen Teams waren darauf angewiesen, ihre Roboter per Hand zu setzen, um so der Selbstlokalisation eine Anfangsposition vorzugeben.

In diesem ersten Test unter realen Bedingungen lernten wir vieles, was wir unter Laborbedingungen nicht einbeziehen konnten. Beispielsweise ist es dringend notwendig, Distanzen einschätzen zu können, um zu wissen, ob der Spieler im Ballbesitz ist, da davon sein Verhalten maßgeblich bestimmt wird. Ein weiterer Faktor für den Erfolg im Spiel ist die Fähigkeit eine Lücke zwischen nahe beieinander stehenden Robotern zu finden, um durch diese zum Beispiel direkt auf das Tor spielen zu können. Zwar funktionierten die Roboter auf den German Open schon mit einer Distanzfunktion (aus Basis einer berechneten e-Funktion), diese war jedoch bedingt durch den Kameraaufbau nur schwer zu kalibrieren und deswegen sehr ungenau. Viele Chancen in den Spielen konnten nicht genutzt werden,

weil der eigene, angreifende Roboter keinen Weg zum Tor finden konnte.

Ebenso sollten Kollisionen weitestgehend vermieden werden, da diese einen klaren Regelverstoß darstellen. Hierzu musste die Hindernisvermeidung verbessert werden. So wurde von Gegnern oftmals nur eine hintere Ecke erkannt, was in Kombination mit der damaligen Distanzeinschätzung den Eindruck hervorrief, der Weg sei frei.

Ein weiteres Problem lag in der Stabilität der Hardware. So stürzten die Kameras aufgrund von Firewireproblemen mehrfach ab und der Kameleon-Motorcontroller offenbarte seine Schwächen. Die betroffenen Roboter mussten kurzzeitig aus dem Spiel herausgenommen werden oder standen für den Rest des Spiels nicht mehr zur Verfügung.

Der Erfolg bei den German Open führte dazu, dass das Team zur Weltmeisterschaft in Padua eingeladen wurde. Es blieben zwei Monate Zeit die bei den German Open erkannten Probleme zu beheben und das Gesamtsystem zu optimieren.

Hierunter fiel die Suche nach einem alternativen Motorcontroller, die Verbesserung der Distanzfunktion, die Verstärkung der Schusseinheit, die Stabilisierung des Firewiresystems der Kamera, die Weiterentwicklung der Selbstlokalisation und des Objekttrackings sowie die Einführung von neuen Strategie-Ansätzen, so genannte „höhere Fähigkeiten“.

Die Umstellung der Systemarchitektur von Threads auf einen serialisierten Ablauf erbrachte eine deutliche Beschleunigung und Stabilisierung der Software. Es konnten in kürzerer Zeit mehr Informationen ausgewertet werden, so dass die Roboter insgesamt schneller und dynamischer wurden.
Als Konsequenz aus den Roboterausfällen wurde zusätzlich der dynamische Rollenwechsel eingeführt, so dass der Platz eines ausgefallenen Roboters von einem anderen übernommen werden konnte.

Der Torhüter, der sich auf den German Open als stark verbesserungsfähig erwies, wurde soweit optimiert, dass viele Teams Interesse am Design bekundeten. Er wurde mit einer neuen Selbstlokalisation ausgestattet, bei der zusätzlich zur normalen Funktionalität noch die Fußpunkte der Torpfosten in die Berechnung einbezogen wurden. Sein Verhalten wurde vollständig neu programmiert, hierunter fallen die Entwicklung eines Beschleunigungsmoduls, um sein Fahrverhalten besser kontrollieren zu können, sowie die Berechnung der Ballgeschwindigkeit. Das Beschleunigungsmodul erwies sich als so vorteilhaft, dass es anschließend in alle Robotertypen integriert wurde.

Einen wesentlichen Beitrag zu den Verbesserungen leisteten die konsequent durchgeführten Systemtests. Die permanente, quantitative Kontrolle der Entwicklung gab Aufschluss über Fortschritte bei den Verbesserungen und die verbleibenden Aufgaben um die Ziele zu erreichen.

Obwohl zwischen German Open und der Weltmeisterschaft auf allen Gebieten deutliche Fortschritte erzielt wurden, zeigte sich doch im Vergleich mit bis dahin unbekannten Teams, dass hinsichtlich Strategie, Endgeschwindigkeit und Ausfallsicherheit der Roboter großes Verbesserungspotential vorhanden ist. In Zukunft sollten daher neben Informatikern auch Maschinenbauer und Elektrotechniker am Vorhaben beteiligt werden.

5.2 Ausblick

Die Projektgruppe hat in der kurzen Zeit von zehn Monaten viel erreicht: sie hat eine Mannschaft von autonomen Fußballrobotern aufgestellt, die auf oberem Niveau im internationalen Vergleich spielten. Software und Hardware weisen viele positive Eigenschaften auf.

An vielen Stellen gibt es Möglichkeiten zur Erweiterung, an denen Weiterarbeit lohnenswert ist. Um das Ziel von schnelleren Robotern zu erreichen, müsste beispielsweise die Übersetzung der Getriebe geändert werden. Damit die Roboter dadurch ihre Orientierungsfähigkeit nicht einbüßen, bliebe zu testen, ob die Frequenz, in der die Applikation arbeitet für eine höhere Geschwindigkeit ausreichend ist.

Anstelle des bisher überwiegend reaktiven Systemverhaltens könnte zukünftig mehr geplant werden. D.h. der Roboter fährt beispielsweise zu einem Punkt, an dem er den Ball im nächsten Zyklus

erwartet. Dies lässt sich durch den fest eingestellten Takt im Programmablauf relativ leicht realisieren.

Die Schusskraft sollte weiter verbessert werden, um auch aus größerer Entfernung auf das Tor schießen zu können.

Weiterer Ansatzpunkt könnte die Verbesserung des Roboterverhaltens durch Einsatz von Lernverfahren sein. Statt Ausprogrammierung des Balldribblings könnte dies gelernt werden. Bisher hat die Mannschaft aus Zeitmangel nur implizites Teamspiel genutzt und auf die Ausnutzung weiterer Möglichkeiten wie die Erstellung eines gemeinsamen Weltbilds verzichtet. Da das bisherige Bildverarbeitungssystem bedingt durch den verwendeten Spiegel, den Ball nicht über das gesamte Spielfeld hinweg verfolgen kann, könnten sich die Roboter gegenseitig mitteilen, wo der Ball gesehen wird. Natürlich bringt ein solcher Ansatz wieder neue Schwierigkeiten mit sich, die gelöst werden müssten: falls die Selbstlokalistion falsch wäre, würde der Ball unter einer falschen Position gesehen werden. Entsprechend würde diese fehlerhafte Information bei der Berechnung der zusammengeführten Ballposition das Ergebnis verfälschen.
Beim Teamspiel selber wäre es schon hilfreich, wenn sich die Roboter gegenseitig mitteilten, dass sie im Ballbesitz sind. In Spielen ließen sich Situationen beobachten, in denen ein Verteidiger im Ballbesitz auf das gegnerische Tor stürmte. Da jedoch der eigentliche Stürmer immer vom Ball angezogen wird, behinderte er den eigenen Roboter beim Attackieren des Tor. Möglicher Lösungsansatz könnte sein, den eigenen Roboter, dessen Position bekannt wäre, durchzulassen. Viele weitere Formen von Teamspiel wären denkbar.

Die Einstellungen der Farbsegmentierung zur Kalibrierung der Bildverarbeitung haben sich für die Zwecke des Robocup als äußerst robust erwiesen. Jedoch ist die Farberkennung noch zu sehr von der Beleuchtung der Umgebung abhängig. Schatten auf den Objekten verfälschen die Farben und können große Probleme bei der Objekterkennung bereiten. Es wäre daher wünschenswert, die Bildverarbeitung so zu verändern, dass in Zukunft markante Regionen im Spielfeld unabhängig von den Beleuchtungsverhältnissen erkannt werden. Möglich wäre es, sich hauptsächlich bei der Orientierung auf Linien im Feld anstelle von farbigen, markanten Objekten zu stützen. Damit würde eine Selbstlokalisation möglich werden, die auf Farbinformationen im Bild fast ganz verzichten könnte.

Nicht zuletzt kann noch einiges im Bereich der Robustheit der Hardware insbesondere in Hinblick auf einen stabileren Kameraaufbau und einen zuverlässigeren Motorcontroller getan werden.
Weitere Erweiterungen im Bereich der Hardware sind denkbar. Beispielsweise könnten taktile Sensoren eingebracht werden, mit denen Kollisionen zuverlässiger erkannt und ein gegenseitiges Festfahren der Roboter vermieden werden könnte

Aus den Erfahrungen, die das Team bei den Weltmeisterschaften gesammelt hat, lässt sich insgesamt sagen, dass mehr Testspiele unter realen Bedingungen abgehalten werden müssen, um wirklich erfolgreich zu sein.

Da das Team seine Stärke auf den Wettbewerben zeigen konnte, hat es einige Einladungen von anderen Teams zu Testspielen erhalten.

Anhang A

Bedienungsanleitung

A.1 Vorwort

Für die Steuerung des Tribots Teams sind zwei Anwendungen von Bedeutung, deren Bedienung in diesem Kapitel beschrieben wird.

Die eigentliche Tribotsanwendung läuft auf jedem einzelnen Roboter und dient der Ansteuerung, TribotsControl ist das Kommunikationsmodul, welches ausgelagert auf einem externen Rechner gestartet wird. Die Entwicklung beider Anwendungen stand neben der erforderlichen Funktionalität unter dem Gesichtspunkt von Softwareergonomität und hoher Benutzerfreundlichkeit. So entstand im Verlauf der Projektgruppe ein Endprodukt, welches sich auszeichnet durch hohe Funktionalität und einfache Bedienung.

Es folgt nun eine ausführliche Erläuterung beider Anwendungen, die einen Überblick über die Funktionen und Konfigurationsmöglichkeiten gibt. Es wird dringend empfohlen, die komplette Anleitung vor dem ersten Einsatz der Roboter zu lesen und sich genau an die hier beschriebenen Vorgehensweisen zu halten. Nur dann ist gewährleistet, dass sich die Roboter fehlerfrei und wie gewünscht auf dem Spielfeld bewegen.

A.2 Tribots-Anwendung

Die Tribots-Anwendung besteht aus folgenden Komponenten:

- Hauptfenster
- Bildverarbeitungsfenster
- Selbstlokalisationsfenster
- Pathfinder-Fenster
- Odometrie-Selbstlokalisationsfenster

A.2.1 Hauptfenster

Das Hauptfenster der Tribots-Anwendung ist in vier Bereiche unterteilt. **Status**, **Active Components**, **GUI Menü** und **Select Parameter**. Abbildung A.1 zeigt das Hauptfenster, das der Anwender nach dem Tribots-Programmstart erhält.

Im **Status**-Bereich werden die grundlegenden Einstellungen bezüglich der Strategie und der Feldaufteilung vorgenommen. Über **Playertype** können dort die unterschiedlichen Rollen für den Roboter ausgewählt werden, zwei Angriffsstrategien, drei Verteidigungsstrategien und verschiedene Torwarttypen. Eine kurze Erläuterung der einzelnen Spielertypen ist in der Kurzreferenz unter A.5.1 zu

finden. Es sollten maximal zwei Angreifer und maximal drei Verteidiger auf dem Feld stehen, da sich bei mehrfacher Zuweisung von Rollen (z.B. zwei Roboter als Attack 1) Strategien überschneiden können. Dennoch ist aber auch eine mehrfache Vergabe möglich. Über den Menüpunkt **Targetgoal** wird die Farbe des gegnerischen Tors und somit die Spielrichtung festgelegt. Angelehnt an das offizielle Robocup Regelwerk stehen hier gelb (**yellow**) und blau (**blue**) als Torfarben zur Auswahl. Eine Übersicht über die angeschlossenen Module und deren Status ist in dem Feld **Active Components** zu finden. Aktive Module werden durch ein grünes Kästchen angezeigt.

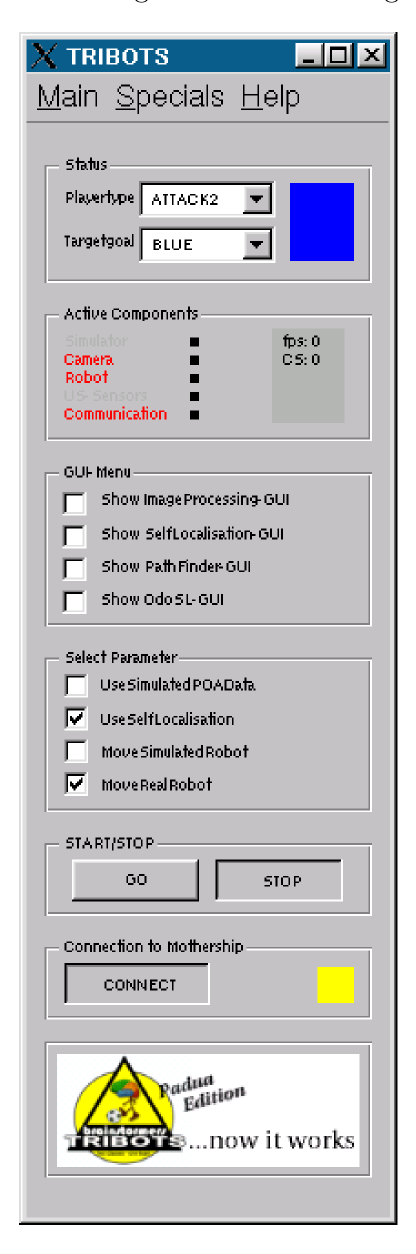


Abbildung A.1: Hauptfenster der Tribots-Anwendung

Im Bereich **GUI Menue** lassen sich über Checkboxen verschiedene Ansichten aktivieren und deaktivieren, die diverse Funktionen und Kalibrationen ermöglichen. Diese werden, wie bereits erwähnt, in den weiteren Kapiteln beschrieben. Über die Sektion **Select Parameter** sind Einstellungen möglich, die insbesondere beim Einsatz eines Simulators von Nutzen sind. Die Parameter haben im einzelnen folgende Bedeutungen:

- **Use Simulated POA Data** Bei aktivierter Checkbox wird ein Possible Object Array ver-

wendet, an Hand dessen der Roboter seinen Fahrweg berechnet. Dieser Punkt simuliert eine künstliche Umgebung. Wichtig beim Einsatz des Simulators.

- **Use Selflocalisation** Über diese Checkbox lässt sich die Selbstlokalisation (de-)aktivieren.
- **Move Simulated Robot** Über diese Checkbox lässt sich die Wiedergabe des eigenen Roboters, der (Spielfeld-)Umgebung und der Fahrbewegungen im Simulator (de-)aktivieren.
- **Move Real Robot** Über diese Checkbox kann man eine Fahrbewegung des Roboters (de-)aktivieren. Dies ist z.B. von Bedeutung, wenn man das Programm laufen lassen möchte, aber nur den Simulator nutzt und den Roboter nicht real fahren lassen möchte.

Die Funktion der Knöpfe im unteren Fensterbereich ist weitesgehend selbsterklärend, deshalb wird an dieser Stelle nur kurz darauf eingegangen: Über **Start / Stop**, wird der eigentlich Fahrmodus des Roboters (de-)aktiviert. **Connect** trennt die Verbindung zu TribotsControl, damit diese dann neu initialisiert werden kann.

Abschließend folgt die Beschreibung des **Fenstermenüs**, teilweise sind die dort enthaltenen Funktionen parallel über die o.g. Knöpfe zu erreichen, der Vollständigkeit halber, aber auch hier zu finden. Abbildung A.2 zeigt die aufgeklappten Kontextmenüs, die man über **Main** und **Specials** erreichen kann. Die Grundfunktionen, die über **Main** zu erreichen sind, sind

- Starten der Module über **Start Main Loop**
- Stoppen der Module über **Stop Main Loop**
- Starten des Roboters über **Go**, Roboter erhält Fahrbefehle
- Notstopp des Roboters über **Stop**, woraufhin keine Fahrbefehle mehr verarbeitet werden
- Homeposition auf dem Spielfeld anfahren über **Go Home**
- Beenden und Verlassen des Programms über **Exit**

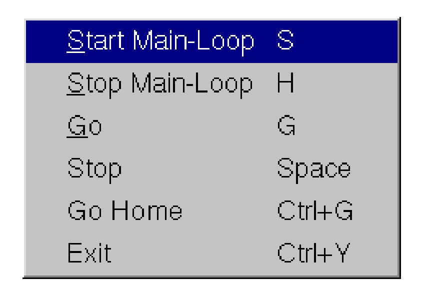


Abbildung A.2: Kontextmenü der Tribots-Anwendung

Zwei Sonderfunktionen, die in der Regel nur selten benötigt werden, aber trotzdem nicht weniger wichtig sind, sind unter dem Menüpunkt **Specials** zu finden. Zum einen kann man hier über 'Distance Calibration' die Funktion kalibrieren, die für die Entfernungsmessung zuständig ist, zum anderen lässt sich hier der Debug Modus aktivieren, über den eine komfortable Fehlererkennung bei Softwareabstürzen o.ä. möglich ist. 'Distance Calibration' muss im Optimalfall für jede Umgebung nur ein einziges Mal durchgeführt werden.

A.2.2 Bildverarbeitungsfenster

Das Bildverarbeitungsfenster dient zwei primären Aufgaben. Zum einen kann man sich jeder Zeit anzeigen lassen, wie gut die Objekterkennung aktuell funktioniert und wie gut die Segmentierung eingestellt ist, zum anderen ist sie das wichtigste Werkzeug zur Kalibrierung der Software. Im Folgenden wird detailliert beschrieben, wie die Kalibrierung vorzunehmen ist, worauf zu achten ist und wie man eine möglichst gute Qualität bezüglich der Bilderkennung erreicht. Abbildung A.3 zeigt zur besseren Verdeutlichung einen Screenshot des Fensters.
Die Bildverarbeitung muss im Robocup die Farben orange (Ball), grün (Spielfeld), gelb (Tor), blau (Tor), schwarz (Hindernisse) und weiß (Linien und Torpfosten) erkennen. Alle sechs Farben können deshalb separat über die graphische Oberfläche eingestellt werden und besitzen hierfür einen eigenen Kalibrierungsbereich mit je drei Schiebereglern.
Neben den Konfigurationsbereichen enthält das Fenster das aktuelle, segmentierte Kamerabild. Auffallend bei dem Bild sind drei Kreise (zwei im äußeren Bereich, einer im inneren Bereich des Bildes), sowie die vom Mittelpunkt des Bildes sternförmig ausgehenden Linien.

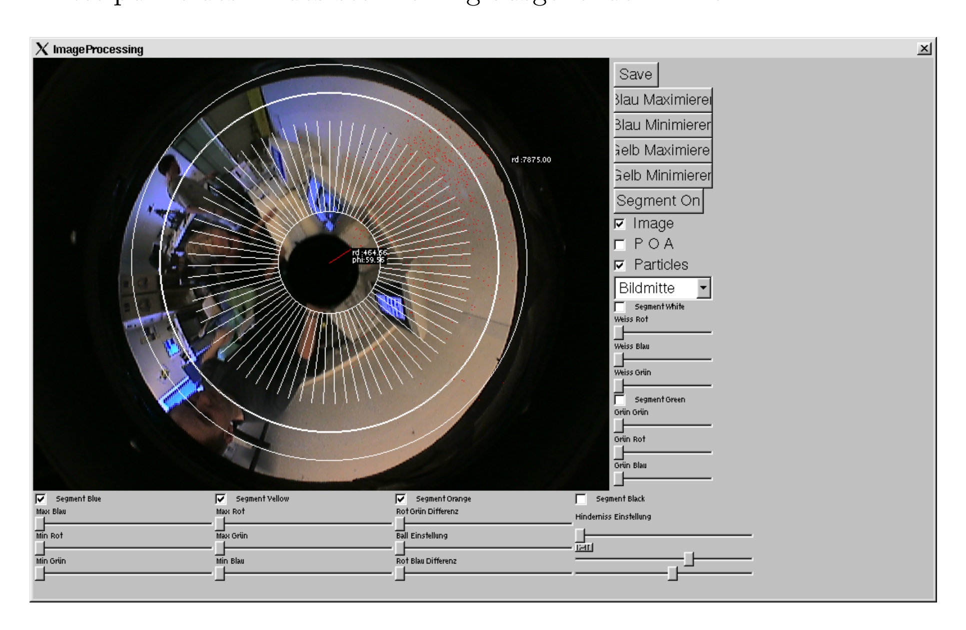


Abbildung A.3: Bildverarbeitungsfenster der Tribots-Anwendung

Alles was sich im inneren Kreis befindet, wird nicht segmentiert, da hier die Bilderkennung zu ungenau ist bzw. das Kameraobjektiv als Hindernis erkannt würde. Der äußere Kreis sollte das obere Drittel der Torpfosten, der mittlere Kreis das untere Drittel der Torpfosten durchlaufen.
Nachfolgend ist nun eine Auflistung der einzelnen Funktionsbereiche zu finden:

- **Save Button** speichert die aktuellen Einstellungen.
- **Segment ON/OFF** schaltet die Anzeige der Segmentierung an / aus.
- **Image** schaltet die Anzeige des Bildes an oder aus.
- **POA** (de-)aktiviert die Anzeige der gefundenen Objekte als Text.
- **Particles** (de-)aktiviert die Darstellung der Partikel- und Geometrieeinstellungen.
- **Segment** Über die Segment-Checkboxen wird die Anzeige der jeweiligen Farbsegmentierung für das Kamerabild ein- bzw. ausgeschaltet.

- **Blau Maximieren** Im Kamerabild kann eine blaue, nichtsegmentierte Fläche mit der Maus eingerahmt werden. Drückt man dann den **Blau Maximieren**-Knopf, wird die dort segmentierte Farbe automatisch für die Blau-Maximierung verwendet. Wird also nur ein Teil des blauen Tors erkannt, kann dieser eingerahmt und der Knopf genutzt werden, um automatisch die Blauerkennung zu verbessern.

- **Blau Minimieren** Im Kamerabild kann eine irrtümlich als blau-segmentierte Fläche eingerahmt werden. Drückt man dann den **Blau Minimieren**-Knopf, wird der erfasste Farbwert für die Blauerkennung reduziert bzw. ausgeschlossen.

- **Gelb Maximieren** Funktion analog zu **Blau Maximieren**

- **Gelb Minimieren** Funktion analog zu **Blau Minimieren**

Offen ist immer noch die Frage, wie das Kamerabild genau segmentiert wird, deshalb folgt nun eine Anleitung für die Kalibrierung mit den vorhandenen Schiebereglern.

Farbeinstellungen Die Farben sollten der Reihe nach kalibriert und nach jeder korrekten Einstellung über den **Save**-Knopf gespeichert werden. Zuerst sollten die Regler so eingestellt werden, dass die Farbe sehr gut erkannt wird, also auch mehr als die gewünschten Objekte. Danach kann über die Regler die Erkennung soweit heruntergedreht werden, dass nur noch die benötigten Objekte segmentiert werden. Im Einzelnen lassen sich die Farben wie folgt segmentieren:

- Blau-Kalibrierung
  Die Regler geben an, wie groß der Blauwert eines Pixels mindestens sein muss, um als Blau erkannt zu werden (**MaxBlau**), wie groß der Rotwert eines Pixels maximal sein darf, um als Blau erkannt zu werden (**MinRot**) und wie groß der Grünwert eines Pixels maximal sein darf, um als Blau erkannt zu werden (**MinGrün**). Am Anfang sollten **MaxBlau** niedrig sowie **MinRot** und **MinGrün** hoch eingestellt sein. Durch Heraufsetzen von **MaxBlau** und Heruntersetzen von **MinRot** und **MinGrün** lässt sich die Erkennung einschränken.

- Gelb-Kalibrierung
  Die Regler geben an, wie groß der Rotwert eines Pixels mindestens sein muss, damit der Pixel als Gelb erkannt wird (**MaxRot**), wie groß der Grünwert eines Pixels mindestens sein muss, damit der Pixel als Gelb erkannt wird (**MaxGrün**) und wie groß der Blauwert eines Pixels maximal sein darf, damit der Pixel als Gelb erkannt wird (**MinBlau**). Am Anfang sollten also **MaxRot** und **MaxGrün** niedrig und **MaxBlau** hoch eingestellt sein. Durch Heraufsetzen von **MaxRot** und **MaxGrün** und Heruntersetzen von **MinBlau** lässt sich die Erkennung einschränken.

- Rot-Kalibrierung
  Die Regler geben an, wie groß die Differenz zwischen dem Rot- und dem Grünwert eines Pixels mindestens sein muss, damit der Pixel als Rot erkannt wird (**RotGrünDifferenz**), wie groß der Rotwert eines Pixels mindestens sein muss, damit der Pixel als Rot erkannt wird (**BallEinstellungen**) und wie groß die Differenz zwischen Rot- und Blauwert eines Pixels sein muss, damit der Pixel als Rot erkannt wird (**RotBlauDifferenz**). Am Anfang sollten also alle Regler niedrig eingestellt sein. Durch Heraufsetzen der Regler lässt sich die Erkennung einschränken.

- Weiß-Kalibrierung
  Die Regler geben an, wie groß der Rotwert eines Pixels mindestens sein muss, damit der Pixel als Weiß erkannt wird (**WeißRot**), wie groß der Grünwert eines Pixels mindestens sein muss, damit der Pixel als Weiß erkannt wird (**WeißGrün**) und wie groß der Blauwert eines Pixels sein

muss, damit der Pixel als Weiß erkannt wird. Am Anfang sollten alle Regler niedrig eingestellt sein. Durch Heraufsetzen der Regler lässt sich die Erkennung einschränken.

- Grün-Kalibrierung
  Die Regler geben an, wie groß der Grünwert eines Pixels mindestens sein muss, damit der Pixel als Grün erkannt wird (**GrünGrün**), wie groß der Rotwert eines Pixels höchstens sein darf, damit der Pixel als Grün erkannt wird (**GrünRot**) und wie groß der Blauwert des Pixels höchstens sein darf, damit der Pixel als Grün erkannt wird (**GrünBlau**). Am Anfang sollte **GrünGrün** niedrig und **GrünRot** und **GrünBlau** hoch eingestellt sein. Durch Heraufsetzen von **GrünGrün** und Herabsetzen von **GrünRot** und **GrünBlau** lässt sich die Erkennung einschränken.

- Schwarz-Kalibrierung
  Für die Schwarz-Kalibrierung ist lediglich ein Regler notwendig, der angibt, wie groß die RGB-Werte eines Pixels höchstens sein dürfen, damit der Pixel als schwarz erkannt wird. Am Anfang sollte der Regler hoch eingestellt sein, durch Herabsetzen kann dann die Erkennung eingeschränkt werden.

A.2.3 Selbstlokalisationsfenster

Abbildung A.4 zeigt das Selbstlokalisationsfenster, welches man über das Hauptfenster wie beschrieben aufrufen kann. In ihm ist eine Übersicht des Spielfeldes zu sehen, auf dem die Partikelverteilung, sowie die wahrscheinliche Position des Roboters (Partikelanhäufung und rotes Fadenkreuz) eingezeichnet ist. Sollten mehrere mögliche Positionen gefunden werden (Selbstlokalisation springt zwischen Positionen hin und her, bzw. schwankt), ist die Selbstlokalisation nicht eindeutig möglich, was unterschiedliche Ursachen haben kann (z.B. fehlerhafte Erkennung der Tore, der Eckfahnen usw.).

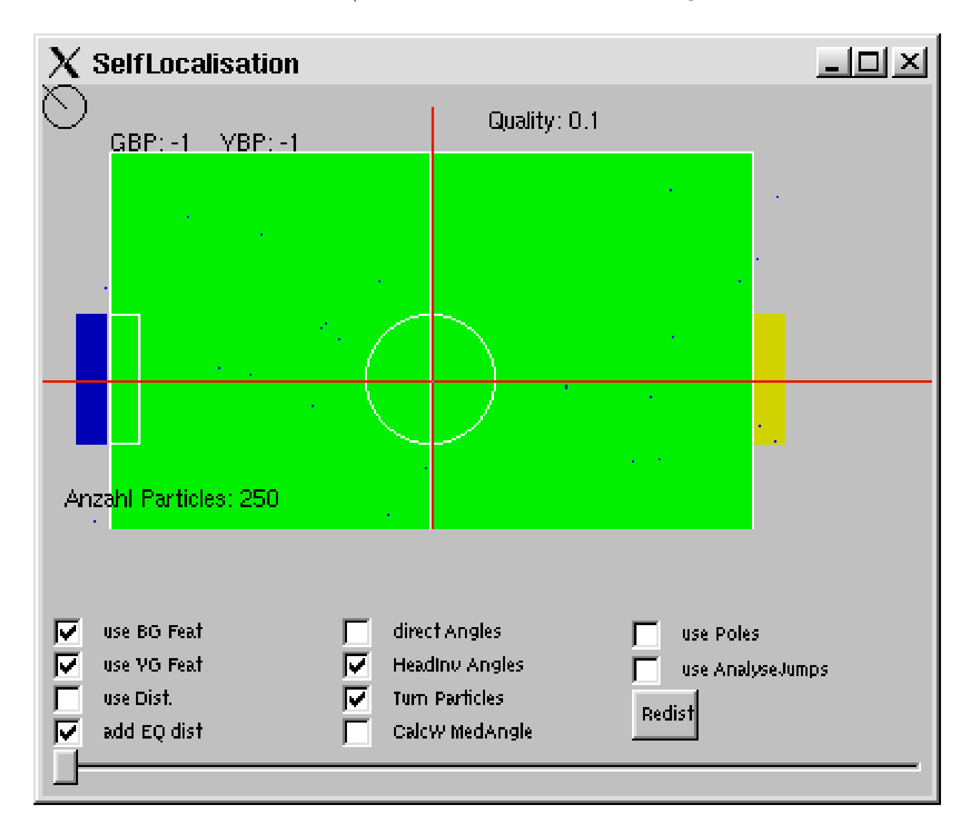


Abbildung A.4: Selbstlokalisationsfenster der Tribots-Anwendung

Über die Checkboxen im unteren Bereich des Fensters lassen sich die folgenden Einstellungen vornehmen:

- **Use BG Feat** das 'Blue Goal Feature' für die Selbstlokalisation (de-)aktivieren
- **Use YG Feat** analoge Bedeutung zu **Use BG Feat**
- **Use Dist**(ance) die Abstände von den Torpfosten fließen in die Selbstlokalisation mit ein
- **Use EQ Dist**(ributed Particles) einstreuen gleichverteilter Partikel, um eine bessere Position zu finden.
- **direct Angles** die Winkel, zur Berechnung der Position, werden direkt in der Bildverarbeitung gemessenen und unabhängig voneinander zur Bewertung der Wahrscheinlichkeit eines Partikels verwendet
- **HeadInv Angles** von der Ausrichtung unabhängige Winkel verwenden (z.B. Winkel zwischen 2 Torpfosten)
- **Turn Particle** die Partikel an den Features ausrichten
- **CalcW MedAngle** die Partikelausrichtung aus dem Mittelwert aller Ausrichtungen berechnen
- **Use Poles** Die Eckfahnen des Spielfeldes für die Selbstlokalisation (de-)aktivieren. Wenn diese z.B. nicht korrekt erkannt werden, würden sie die Selbstlokalisation verfälschen und können deshalb deaktiviert werden.
- **Use Analyse Jumps** Herausfinden, ob die Selbstlokalisation stark schwankt und in diesem Fall automatisch ganz auf Odometrie umschalten. *Achtung*: diese Funktion funktioniert auf dem Feld ganz gut, im Tor aber so gut wie gar nicht.

Als weitere Funktion, die die Oberfläche der Selbstlokalisation bietet, kann man über den Regler im unteren Bereich des Fensters, ein aufgezeichnetes Possible Object Array vor- bzw. zurückzuspulen. Dies kann verwendet werden, um die Selbstlokalisation unabhängig von Hardware zu testen.

A.2.4 Pathfinder-Fenster

Das Pathfinder-Fenster bietet die Möglichkeit, die Richtungsvektoren der Potentialfeld Methode sichtbar zu machen und so die Fahrtplanung des Roboters kontrollieren, bzw. nachvollziehen zu können. Wie in Abbildung A.5 zu sehen ist, setzt sich das Fenster aus vier Komponenten zusammen, dem **Info-Fenster**, dem **Tool-Fenster**, dem **Replay-Fenster** und der **strategischen Spielfeldansicht**.

Das **Info-Fenster** enthält verschiedene Daten, die aktuell ausgelesen werden. Hierzu gehört die aktuelle Position des Roboters (0/0=Mittelpunkt des Spielfeldes), die aktuelle Geschwindigkeit in $\frac{m}{s}$, die Fahrbefehle, die an die Motoren gehen und schließlich der Movemode. Movemode bedeutet, in welcher strategischen Situation sich der Roboter aufgrund der vorliegenden Daten gerade befindet. Der Roboter könnte sich z.B. in der Situation `kein Ballbesitz - sehe Ball - Ball im Aktionsbereich - fahre zu Ball` befinden.

Im **Tool-Fenster** ist die Steuerung der Log-Datei möglich. Über **Replay** wird automatisch die temporäre Log-Datei geladen. Der Datenmitschnitt läuft, solange im Hauptfenster **Go** aktiviert ist. **Clr Log** leert die temporäre Log-Datei wieder und über **ld Log** kann eine externe Log-Datei geladen werden, die aber nicht größer sein sollte als ca. 15MB. Die Checkbox **vec** aktiviert, bzw. deaktiviert die Ansicht der Vektoren.

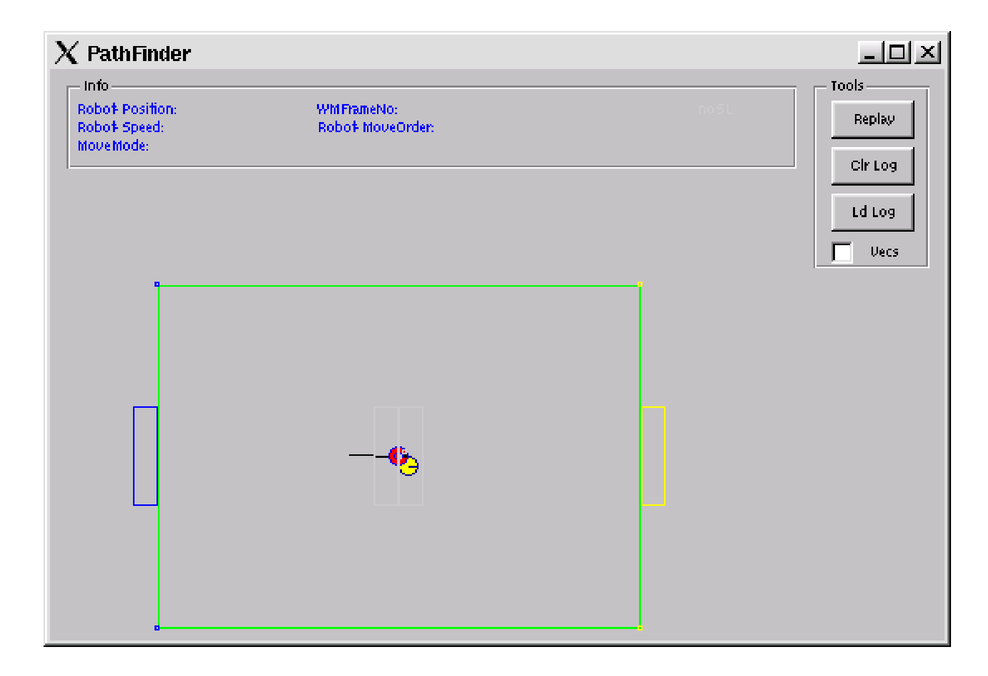


Abbildung A.5: Pathfinder-Fenster der Tribots-Anwendung

Der **Replay-Bereich** des Fensters enthält alle Funktionen für die Wiedergabe einer aufgezeichneten Log-Datei. Sowohl über den Schieberegler als auch über die entsprechenden Knöpfe lässt sich Vor- und Zurückspulen, die Wiedergabe stoppen oder im Normalmodus abspielen. Wiedergegeben wird das über den **Replay**-Knopf im Tool-Fenster aufgenommene Verhalten des Roboters bzw. das der geladenen Log-Datei. Diese Funktion bietet gute Möglichkeiten für die Analyse des Fahrverhaltens des Roboters.

Die **strategische Spielfeldansicht** stellt schließlich alle aufbereiteten Daten grafisch dar: den eigenen Roboter nebst Ausrichtung (grauer Kreis mit schwarzer Linie), den Ball (rot gefüllter Kreis), das momentane Ziel mit dem dazugehörigen Einflussradius (blauer Kreis), alle Hindernisse (schwarze Kreise), natürlich das Spielfeld, beide Tore und - wenn aktiviert - die Vektoren. Zu beachten ist, dass das momentane Ziel vom Ball abweichen kann, wenn der Ball z.B. nicht gesehen wird und deshalb eine alternative Position angesteuert wird.

A.2.5 Odometrie-Selbstlokalisationsfenster

Das OdoSL-Fenster dient der Kontrolle der Selbstlokalisation, die auf den Odometriedaten basiert. Abbildung A.6 zeigt einen Screenshot des Fensters.

Der Roboter wird in die Mitte des Spielfeldes gesetzt und über den Knopf **Or Click Here** die Selbstlokalisation anschließend initialisiert. Auf der Spielfeldübersicht wird daraufhin die berechnete Position des Roboters eingezeichnet und kann während der Fahrt überwacht werden. Das Fenster enthält zusätzlich alle Daten der Selbstlokalisation und die Ansteuerungsbefehle der Motoren (unten links im Bild).

A.3 TribotsControl-Anwendung

A.3.1 Allgemeine Informationen

Die Kommunikationszentrale des Teams *Tribots - Brainstormers* bildet das Programm *TCPE (TribotsControl Padua Edition)*. Hier laufen während des Spiels alle von den Robotern per WLAN über-

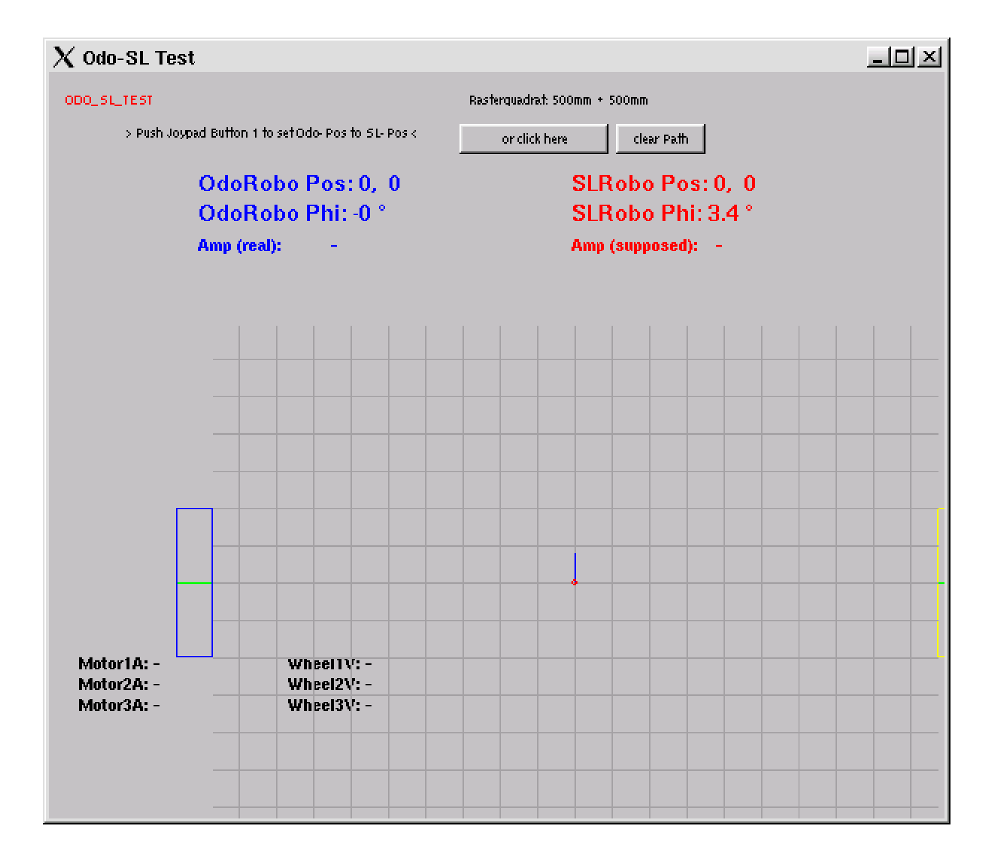


Abbildung A.6: Odometrie-Selbstlokalisationsfenster der Tribots-Anwendung

tragenen Informationen zusammen und geben dem Anwender die Möglichkeit, die aktuelle Rolle des jeweiligen Roboters (Goalie, Defender oder Attacker) und seine Zielfarbe (hiermit ist die Farbe der Gegenspieler und deren Torfarbe gemeint) abzulesen. Weiterführend kann der Anwender mit der Kontrolloberfläche dem Roboter rudimentäre Kommandos geben, die später noch näher erläutert werden.

Nachdem das Programm über die Shell mit dem Befehl **tcpe** gestartet wurde, erhält der Anwender den Startbildschirm (vgl. Abbildung A.7). Deutlich zu erkennen sind die 5 verschiedenen Teilbereiche der Oberfläche. In der oberen Hälfte des Bildschirms befinden sich die einzelnen Controlpanels für die jeweiligen Roboter. Die untere Hälfte ist für allgemeine Befehle und Informationen aller vier Spieler reserviert.

A.3.2 Kommunikation mit einzelnen Robotern

Jedem Roboter steht ein **Messagefield** zur Verfügung, indem allgemeine Statusmeldungen des jeweiligen Spielers angezeigt werden können. Es wird ausgegeben, ob der Spieler sich verbunden hat, in welchem Modus er sich befindet oder ob er noch mit dem TCPE in Verbindung steht. Statusmeldungen innerhalb dieses Feldes können u.a. **Roboter connected** oder **connection closed** sein. Hinzu kommen Meldungen bei Rollenwechseln, z.B. **new Role: DEFEND2** oder auch Fehlermeldungen wie z.B. **Failure Report - Worldmodel Crash**.

Unterhalb des Messagefield sind 4 Signalanzeigen zu sehen, die den Status der vier wichtigsten Funktionen des Roboters anzeigen. Das Signal **WLAN** zeigt eine vorhandene WLAN-Verbindung an, falls es grün leuchtet und eine unterbrochene oder nicht vorhandene, wenn es rot markiert ist. Mit **outgoing communication** und **incoming communication** wird der Datenverkehr mit dem Roboter und von eben diesem angezeigt. Das grüne Blinken stellt die eingehenden bzw. ausgehenden

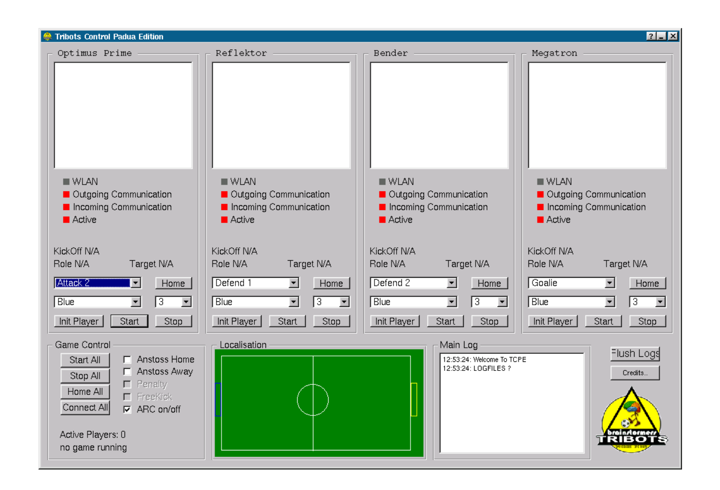


Abbildung A.7: Graphische Oberfläche von TCPE

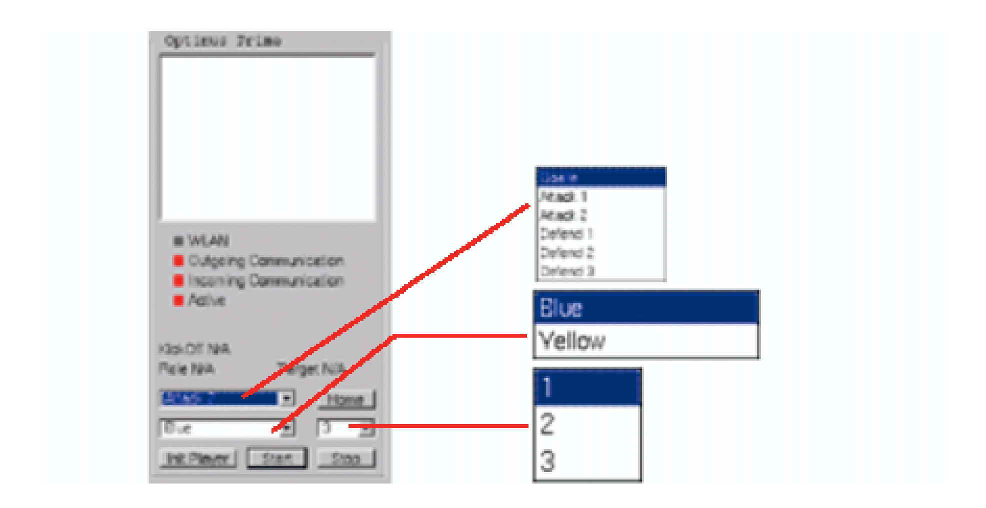


Abbildung A.8: Player Panel von TCPE

Datenpakete innerhalb der Kommunikation mit dem Roboter dar. Das Signal **Active** leuchtet grün, falls der Roboter eingeschaltet und betriebsbereit sein sollte, rot, wenn dieser Zustand nicht erreicht wurde.

Die drei Felder **KickOff N/A**, **Role N/A** und **Target N/A** stellen den aktuellen Status des jeweiligen Roboters dar. Dazu gehört, ob er im Anstoß-Modus ist, welche Rolle er innerhalb des Teams zur Zeit spielt und welche Farbe seine Zielfarbe (also die Farbe seiner Gegner und somit auch des Tors auf das er spielen soll) ist. Unter dieser Anzeige kann der Anwender das Verhalten des Roboters umstellen. Hier stehen ihm verschiedene Rollen zur Verfügung, die der Roboter innerhalb des Teams einnehmen kann (Goalie, Attack, Defender). Auch die Target-Color kann hier eingestellt bzw. geändert werden.
Das Pulldownmenue mit den Werten 1 . . . 3 bietet dem Anwender die Möglichkeit, die **KickOff Preset** umzustellen. Je nach Wert führt der Spieler den eigenen Anstoß auf verschiedene Art und Weise aus. Der Wert **1** bedeutet zum Beispiel, dass der Roboter, der den Anstoß durchführen, mit dem Ball direkt zum Angriff übergehen soll, wogegen **2** und **3** dem Roboter sagen, dass er den Ball vorher noch einem Mitspieler zupassen muss.
Die so geänderte Konfiguration muss mit dem Knopf **Init Player** zum Roboter geschickt werden, so dass dieser die Einstellungen übernehmen kann. Mit den Knöpfen **Start** und **Stop** kann der jeweilige

Roboter gestartet bzw. gestoppt werden.

A.3.3 Kommunikation mit dem Team

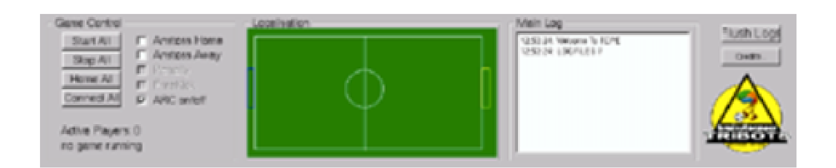


Abbildung A.9: Team Panel von TCPE

Im unteren Bereich des Hauptfensters liegt das Controlpanel für alle Roboter. Hier kann der Anwender allgemeine Befehle an alle Roboter zusammen schicken und somit das Team steuern.
Zunächst einmal hat er vier Knöpfe auf der linken Seite zur Verfügung, mit denen er das Spiel starten (**Start All**), das Team stoppen (**Stop All**) oder alle Roboter auf Ihre vordefinierten Basispositionen setzen kann (**Home All**).

Mit **Connect all** wird die Verbindung zwischen allen Robotern und dem TCPE - Hauptprogramm hergestellt. Fehler und Probleme werden im Fenster **Main-Log** auf der linken Seite ausgegeben. Die Anzeige **Active Player** zeigt die Anzahl der spielbereiten, aktiven Spieler an, und **no game running** besagt, dass zur Zeit kein Spiel stattfindet. Des weiteren hat der Anwender unter **Game Control** die Möglichkeit, in den Checkboxen **Anstoss Home**, **Anstoss Away** und **ARC on/off** allgemeine Einstellungen vorzunehmen, die das gesamte Spiel, das gesamte Team betreffen. So kann der Anwender mit **Anstoss Home** dem Team signalisieren, dass das eigene Team Anstoß hat und das Spiel beginnen darf. Hingegen hat der Gegner Anstoß, wenn **Anstoss Away** aktiviert ist. Der Haken bei **ARC on/off** dient dem (De-)aktivieren des **Automatic Role Change**, der automatischen Rollenverteilung im Spiel. Wenn diese aktiviert ist, übernehmen die Spieler während des Spiels autonom andere Rollen, falls diese durch Ausfall des verantwortlichen Roboters nicht besetzt sein sollten. Diese Haken müssen gesetzt sein, bevor **connect all** und **start all** betätigt werden.

In dem Spielfeld unter **Localisation** werden grafisch die aktuellen, von der Selbstlokalisation der Roboter errechneten, Positionen der einzelnen Spieler angezeigt. So hat der Anwender jederzeit die Möglichkeit, das Verhalten und die Weltsicht der Teamspieler zu erkennen und Fehlverhalten dementsprechend unter Verwendung der mitgeschriebenen Logfiles zu interpretieren. Diese Logfiles können mit dem Knopf **Flush Logs** auf der rechten Seite entleert werden, da diese im Laufe der Zeit sehr groß werden können.

A.3.4 Bedienung von TCPE

Wie schon erläutert, bietet TCPE dem Anwender die Möglichkeit, sowohl einzelne Roboter zu starten, als auch das gesamte Team zu aktivieren. Um ein Spiel zu beginnen muss der Anwender auf **connect all** drücken, um eine Verbindung mit allen Robotern auf dem Feld herzustellen. Sollte es bei einem oder mehreren Robotern auf Anhieb nicht gelingen, so muss man den Knopf **connect all** solange nochmal betätigen, bis die Verbindung zu allen Robotern besteht. Erst dann kann man den Befehl **start all** oder auch **stop all** betätigen, womit erst jetzt das Spiel gestartet und die **ARC** aktiviert wird. Sollte man nur einen einzelnen Roboter steuern wollen, kann man dieses über die einzelnen Controlpanel im oberen Bereich des Bildschirms machen. Auch hier gelten die selben Bedienregeln wie für das gesamte Team. Zuerst muss man **init player** drücken, um den Spieler zu aktivieren und ihn dann per **start** und **stop** steuern zu können.

A.4 Tribots-Programmstart

Der Übersicht halber wird an dieser Stelle nochmal eine Zusammenfassung gegeben, welche Schritte notwendig sind, um den Roboter korrekt fahren zu lassen. Der nachfolgende Beispielablauf ist eine minimale Anleitung, um den Roboter zum Laufen zu bekommen.

1. Tribots-Anwendung starten
   Die Tribots-Anwendung wird auf dem Laptop des Roboters aus dem Tribots-Verzeichnis über `tribots` gestartet.

2. Programm starten
   Die Hardware-Komponenten sollten nun mit **s** gestartet werden. Die Anwendung wartet nun auf die entsprechenden Befehle der Kommunikationssoftware.

3. Module prüfen
   In dem Feld **Active Components** sollten nun alle wesentlichen Module aktiv, also mit einer grünen Markierung versehen sein. Ist dies nicht der Fall, liegt ein Fehler vor, eventuell muss z.B. die Kamera neu gestartet werden. Aktiv sein sollten unbedingt **Camera**, **Communication** und **Robot**.

4. Spielfeldeinstellungen vornehmen
   Über **Status** muss nun die Farbe des gegnerischen Tors und die Rolle des eigenen Roboters ausgewählt werden.

5. Farbsegmentierung überprüfen
   Mit der Taste **i** muss nun die Bildverarbeitungsoberfläche gestartet werden. Es ist vor jedem Start unbedingt erforderlich, dass hier die Objekterkennung überprüft und eventuell die Farbsegmentierung korrigiert wird. Das Fenster kann anschließend wieder mit **i** geschlossen werden.

6. Selbstlokalisation überprüfen
   Um böse Überraschungen nach dem Start des Roboters zu vermeiden, ist es ratsam auch noch einen Blick auf die Selbstlokalisation zu werfen. Wird die Position des Roboters dort nicht einwandfrei erkannt, liegt das in der Regel an einer mangelhaften Einstellung der Bildverarbeitung, also zurück zu Punkt 5.

7. Fahrbefehle starten
   Über **Go** bzw. über die Kommunikationssoftware die Fahrbefehle aktivieren.

Bei Problem mit der Kamera bitte beachten: Eines der wenigen Probleme, die hierbei auftreten können, ist, dass die Kamera nicht gefunden wird (kein grünes Signal hinter dem Kamera Modul) oder abstürzt. In beiden Fällen sollte die Tribots-Anwendung beendet, die Stromzufuhr der Kamera kurz unterbrochen und `sudo firewire restart` ausgeführt werden. Die Tribots-Anwendung anschließend wieder starten, im Normalfall sollte die Kamera dann korrekt arbeiten.

A.5 Kurzreferenzen

A.5.1 Playertypen

- Goalie
  Dies ist der 'Paderborn'-Torwart, mäßig stark, aber immer für Überraschungen gut.
- Attack1
  Der Angreifer darf sich ausschließlich im gegnerischen Spielfeld aufhalten und wartet dort, bis er in Ballbesitz ist / in Ballbesitz gelangen kann.
- Attack2
  Der Angreifer darf sich in beiden Spielfeldhälften bewegen und steuert bei Ball-Besitz umgehend das gegnerische Tor an.
- Defend1
  Der Verteidiger deckt die rechte Seite vor dem eigenen Tor ab.
- Defend2
  Der Verteidiger deckt die linke Seite vor dem eigenen Tor ab.
- Defend3
  Der Verteidiger darf beide Seiten abdecken / befahren.
- GoalieNew
  Weiterentwicklung des 'Paderborn'-Torwarts, nicht perfekt, er verfügt über eine bessere Selbstlokalisation.
- GoalieNewRel
  Die Torwart entspricht weitestgehend dem GoalieNew, mit dem Unterschied, dass dieser relativ fährt, d.h. er muss beim Start genau mittig vor dem Tor platziert werden.
- GoalieYMove
  Der Torwart bleibt auf einer Linie vor seinem Tor und bewegt sich ausschließlich in y-Richtung, also links-rechts.
- GoalieMixed
  Erste Version des 'Padua'-Torwarts, mit guter Selbstlokalisation und (fast) perfektem Stellungsspiel.

A.5.2 Direktwahl-Tasten

Taste	Bedeutung	Taste	Bedeutung
i	Aufruf Bildverarbeitungsfensters	l	Aufruf Selbstlokalisationsfensters
p	Aufruf Pathfinder-Fensters	o	Aufruf Odo-SL-Fensters
s	Starten der Main Loop	h	Stoppen der Main Loop
g	Starten des Roboters		
CTRL+g	Homepositionen anfahren	CTRL+y	Beenden der Tribots-Anwendung
SPACE	sofortiges Stoppen des Roboters		

Tabelle A.1: Direktwahltasten in der Tribots-Anwendung

A.6 Problembehebung

Dieser Abschnitt beschreibt mögliche Probleme beim Betrieb des Roboters und Bedienungsfehler des Tribots-Programms und deren Lösung. Natürlich können die beschriebenen Probleme auch vielfältige andere Ursachen haben, hier sollen aber nur ein paar Ursachen herausgestellt werden, die im praktischen Betrieb der Roboter aufgefallen sind.

- Problem: Durch Starten der Komponenten (**s**) wird die Kamera nicht gestartet

  mögliche Ursache: Kamera-Akku leer
  Lösung: Akkus überprüfen und ggf. wechseln

  mögliche Ursache: Kabelverbindungen zur Kamera locker
  Lösung: Firewire Kabel überprüfen

  mögliche Ursache: Kamera wurde nicht korrekt initialisiert
  Lösung: Programm beenden und per `sudo firewire restart` Kameramodule neustarten, danach Tribots-Programm erneut aufrufen und die Komponenten starten (**s**). Tritt das Problem erneut auf, die Stromzufuhr der Kamera kurz unterbrechen (per Schalter am Roboter) und erneut starten.

- Problem: Starten der Komponenten (**s**) schlägt fehl, es liegt aber kein Kameraproblem vor

  mögliche Ursache: ein Akku ist leer oder schwach
  Lösung: alle Akkus überprüfen und ggf. wechseln

  mögliche Ursache: Board ist aus
  Lösung: alle Schalter am Roboter einschalten (Netz oder Batteriebetrieb)

- Problem: Trotz laufendem Programm bewegt sich der Roboter nicht

  mögliche Ursache: viele Ursachen möglich
  Lösung: bewegt sich der Roboter wenn man auf **Go** klickt, liegt der Fehler möglicherweise an der Kommunikation (z.B. W-LAN). Tut er dies nicht, alle Akkus prüfen, ggf. das Programm sowie Board und Kamera neustarten.

- Problem: Roboter bewegt sich seltsam, tendiert dabei zum gelben Tor

  mögliche Ursache: Im gelben Tor wird rot segmentiert, dies wird aber durch das Gelb verdeckt
  Lösung: Im Bildverarbeitungsfenster (**i**) die Rot-Segmentierung ein-, aber die Gelb-Segmentierung ausschalten. Sieht man jetzt rot segmentierte Pixel im Tor, diese herunterregeln, bis nur noch der Ball oder zumindest (möglichst) viel vom Ball, aber keine (roten) Pixel im gelben Tor segmentiert werden.

- Problem: Roboter bewegt sich ruckartig über das Spielfeld

  mögliche Ursachen: Es werden gelbe Pixel im Spielfeld erkannt
  Lösung: Die Gelb-Segmentierung an mehreren Stellen auf dem Spielfeld überprüfen

- Problem: Roboter fährt sehr häufig zum Spielfeldrand, auch wenn sich dort kein Ball befindet

  mögliche Ursache: Neben dem Spielfeld befinden sich Gegenstände mit gleicher oder ähnlicher Farbe wie der Ball
  Lösung: Rotsegmentierung überprüfen und wenn möglich Gegenstände entfernen

- Problem: Die Schusseinheit löst nicht an der Position aus, wo sie soll / Roboter lokalisieren sich sehr schlecht

  mögliche Ursache: neben dem Spielfeldrand sind Gegenstände, die als Tore erkannt werden
  Lösung: Blausegmentierung überprüfen und wenn möglich Gegenstände entfernen

  mögliche Ursache: Blaue Banderolen (der Gegner) werden als Tor erkannt
  Lösung: Die Blausegmentierung mit einer Banderole im Sichtbereich des Roboters justieren, dabei die Banderole bewegen um verschiedene Blautöne zu überprüfen

- Problem: Roboter kollidiert sehr häufig mit Hindernissen

  mögliche Ursache: Hindernisseinstellung fehlerhaft
  Lösung: Die Hindernisseinstellung im Bildverarbeitungsfenster höher stellen

  mögliche Ursache: Banderolen werden als Ball erkannt
  Lösung: Wie oben beschrieben die Rotsegmentierung justieren

- Problem: Roboter schießt ein Tor nach dem anderen

  mögliche Ursache: Alles perfekt eingestellt
  Lösung: Sekt kaltstellen.

Anhang B

Hardware

In diesem Kapitel finden sich detaillierte Informationen zur Hardware wieder, um einen tiefergehenden Eindruck des Zusammenwirkens der in Kapitel 2 vorgestellten Komponenten bieten.

B.1 Projektion eines Weltpunktes auf einen Bildpunkt

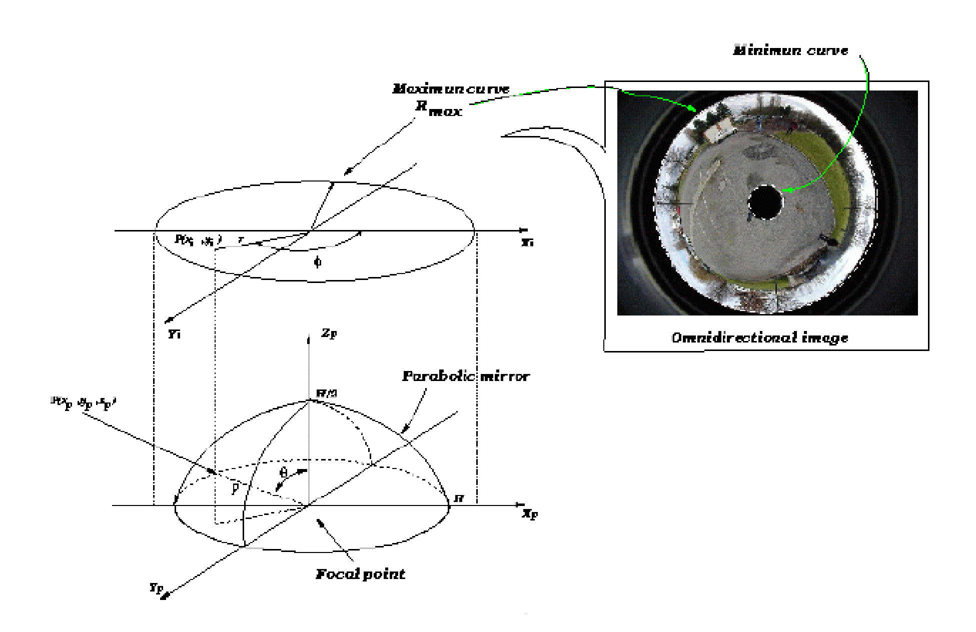


Abbildung B.1: Projektion eines Weltpunktes auf einen Bildpunkt

Um ein perspektivisches Bild mit Hilfe eines omnidirektionalen Kamerasystems zu erstellen, muss eine Beziehung zwischen einem Weltpunkt und einem aufgenommenen Bildpunkt der Kamera erstellt werden. So wird der Anwender in die Lage versetzt, ein Panoramabild in ein perspektivisches Bild zu transformieren.

Für dieses Problemstellung kann man die klassischen Bildverarbeitungsmethoden anwenden.
Der parabolische Spiegel stellt die x_p/y_p Ebene dar, bei x_i/y_i Ebene handelt es sich um das aufzunehmende Bild. Durch trigonometrische Gleichungen rechtwinkliger Dreiecke kann man eine Beziehung zwischen Weltpunkt p und Bildpunkt p' herstellen. Als Ergebnis dieser Umformungen erhält man aus dem Panoramabild ein perspektivisch korrektes Bild.

Wie in Abbildung B.1 zu sehen, wird ein Zylinder dargestellt, der als geometrisches Grundgerüst die Projektion wiedergeben soll. In der x_p/y_p-Ebene wird der parabolische Spiegel und parallel zu ihm

in der x_i/y_i-Ebene die Kamera, genauer das aufgenommene Bild dargestellt. Die beiden abgebildeten Winkel θ und ϕ können mit Hilfe von geometrischen Formeln umschrieben werden.

$$\theta = \cos^{-1}\left(\frac{Z_p}{\sqrt{x_p^2 + y_p^2 + z_p^2}}\right) \quad (B.1)$$

$$\phi = \tan^{-1}\left(\frac{y_p}{x_p}\right) \quad (B.2)$$

Nach einige Umformungen erhält man als Werte für x_i und y_i:

$$x_i = p \cdot \sin(\theta) \cdot \cos(\phi) \quad (B.3)$$

$$y_i = p \cdot \sin(\theta) \cdot \sin(\phi) \quad (B.4)$$

$$\text{wobei } p = \frac{h}{(1 + \cos\phi)}$$

Anhand dieser Formel erhält man den Bildpunkt p' mit den Komponenten x_i und y_i. Durch eine Projektion des sin bzw. cos auf die Koordinatenachsen erhält man Formel B.3.

Im ersten Schritt wird mit $sin\,(\theta)$ die Skalierung p in die x_p/y_p-Ebene projiziert. Im zweiten Schritt wird mit $cos\,(\phi)$ bzw. $sin\,(\phi)$ die x_i- bzw. y_i-Komponente des aufgenommenen Bildes bestimmt. Hierdurch erhält man letztendlich eine Beziehung zwischen einem Weltpunkt und einem Bildpunkt.

B.2 Anschlussplan

Wie schon in Kapitel 2 erwähnt, besteht die Roboterelektronik im Wesentlichen aus dem Kamerasystem, dem Steuerrechner und dem Motorcontroller. Hinzu kommt z.B. die Stromversorgung für die Firewire-Kamera durch einen Firewire-Repeater mit eingeschleifter Stromversorgung und die Ansteuerung der Schusseinheit.

Einige der Komponentenverbindungen weichen bei den verschiedenen Motorcontrollern ab und werden nachfolgend gesondert dargestellt, zuerst jedoch arbeiten wir die Gemeinsamkeiten heraus.

Die Firewire-Kamera DFW-V500 von Sony[11] wird normalerweise mittels Firewirebus durch den Host mit Strom versorgt. Nach dem IEEE1394-Standard wird bei mobilen Endgeräten davon abgesehen, da die Akku-Laufzeit ebendieser stark darunter leiden würde. Der benötigte Strom muss somit aus einer externen Quelle in die Kamera gespeist werden - allerdings ist dieses kameraseitig eigentlich nicht vorgesehen, es existiert z. B. keine Buchse für eine externe Stromversorgung. Unsere Lösung liegt im Einsatz eines Firewire-Repeaters, über den die Stromversorgung der Kamera läuft. Der Repeater wird einerseits durch ein 4-Pol/6-Pol - Kabel mit dem mobilen Rechner verbunden und auf der anderen Seite durch einen 6-Pol/6-Pol - Kabel mit der Kamera. Als externe Stromquelle wird der 12 V-Akkupack mit in den Repeater eingeschleift, die Kabel dorthin sind in den Abbildungen B.2 und B.4 angedeutet.

Die Kommunikation der Roboter mit dem Hostrechner (vgl. Abschnitt 3.5) läuft über eine WLAN-Verbindung ab, ein WLAN-USB-Adapter übernimmt diese Aufgabe. Die Stromversorgung des Adapters erfolgt über das USB-Kabel, seine Übertragungsrate entspricht den bis dato üblichen 11 MBit pro Sekunde, neuere den IEEE802.11g Standard unterstützende Geräte mit 54 MBit pro Sekunde sind für unseren Zweck überdimensioniert. Wesentliches Entscheidungskriterium für diesen Adapter ist die WiFi-Kompatibilität, d.h. Geräte verschiedener Hersteller arbeiten ohne Komplikationen zusammen.

Zeitweise wurde der Einsatz einer WLAN-Bridge getestet, da sie eine größere Sendeleistung besitzt. Ein essentieller Unterschied in der Signalqualität konnte von unserer Seite nicht festgestellt

werden, weiterhin benötigt die Bridge eine zusätzliche Stromversorgung und ist wesentlich größer als das USB-Pendant, weshalb wir uns für die USB-Lösung entschieden haben.

Das eingesetzte Subnotebook der Marke JVC besitzt keine serielle Schnittstelle nach dem RS232-Standard. Der Nachrichtenaustausch mit dem Motorcontroller erfolgt über einen PCMCIA-2-Serial-Adapter, der an dem Laptop angeschlossen ist. Zwischen Kontroller und Adapter verläuft eine 9-poliges serielles Kabel (1-1-Belegung). Eine Lösung durch einen USB-2-Serial-Adapter wurde verworfen, da die benötigte Echtzeitkommunikation nicht realisiert werden konnte.
Beachte: Bei der seriellen Kommunikation über den PCMCIA-Adapter wird ein Stopbit verschluckt.
Beachte: Der Adapter besitzt einen Stromsparmodus und wechselt in einen Ruhemodus, sofern längere Zeit keine Daten über ihn verschickt wurden. Ist er in diesem Ruhezustand und erhält Datenpakete, so verschluckt er die ersten während der Rückkehr in den aktiven Modus.

B.2.1 Kameleon

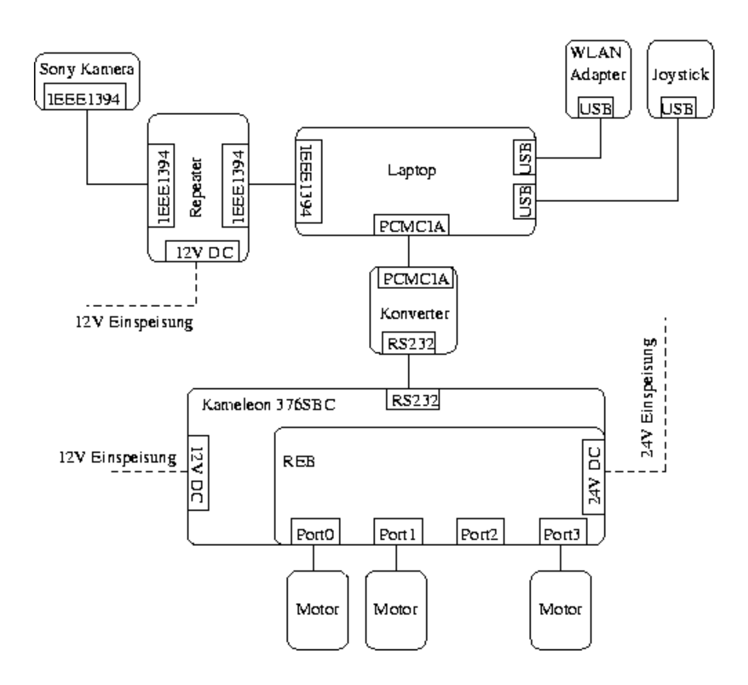


Abbildung B.2: Datenkabel, Konverter und Adapter zwischen den Roboterkomponenten (Kameleon)

Den Motorcontroller K376SBC speist ein separater Akkupack mit 12 V DC, die Verbindung zwischen ihnen ist über ein AMP-Steckersystem aus dem Modellbau-Bereich realisiert. Dieses bietet einen Verpolungsschutz. Für den Betrieb des REB[1] existiert ein separater Stromkreis, Elektronik- und Motorkreis sind damit getrennt, ein Rückfließen von Motorströmen in die Elektronik ausgeschlossen. Zwei 12 V DC Akkupacks, über das AMP-Steckersystem in Reihe geschaltet, genügen dem REB, um die drei Motoren anzutreiben.

Die Massen der zwei verschiedenen Stromkreise sind intern auf dem Kameleonboard zusammengeschalten.

Ein 10-adriges Flachbandkabel verläuft zwischen REB und jedem angeschlossenen Motor. Je drei Adern liegen auf +12 V DC bzw. Masse und sind direkt mit den Motorpolen verbunden, die restlichen vier Adern mit dem Motorencoder. Somit ist eine Regelung möglich. Über diese vier Kanäle werden

[1] Robotic Extension Board

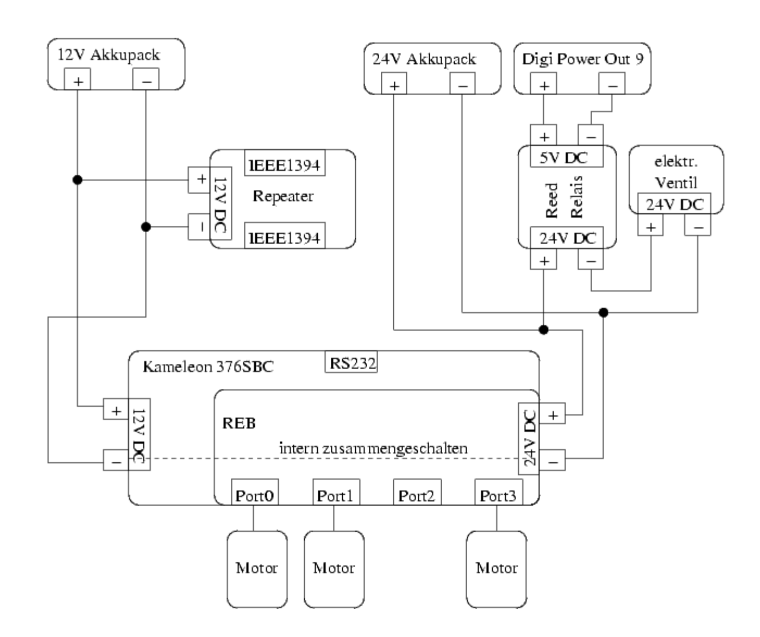


Abbildung B.3: Stromkabel zwischen den Roboterkomponenten (Kameleon)

Informationen über die Radbewegung bzw. deren Inkremente übertragen, für die genaue Verschaltung siehe [8, 9].

B.2.2 TMC

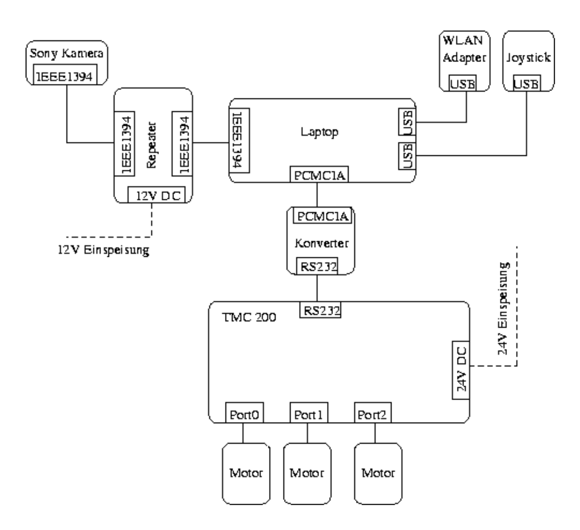


Abbildung B.4: Datenkabel, Konverter und Adapter zwischen den Roboterkomponenten (TMC)

Anders als beim K376SBC arbeitet das TMC200 nur mit einer einzigen Versorgungsspannung von 24 V DC (siehe Abbildung B.5). Auf dem Roboter gibt es damit zwei Spannungskreise, deren Massen nicht verbunden sind. Daraus entstehen u.U. Massendifferenzen zwischen den beiden Kreisen und somit ein nichtbeabsichtigter Stromfluß. Zur Lösung dieses Problems wurden die Massen zusammen auf ein gemeinsames Potential „gezogen“.

Daten- und Stromleitungen der Motoren werden strikt voneinander getrennt, d.h. die bisherige Lösung eines Flachbandkabels wie beim K376SBC muss modifiziert werden. Die Stromversorgung

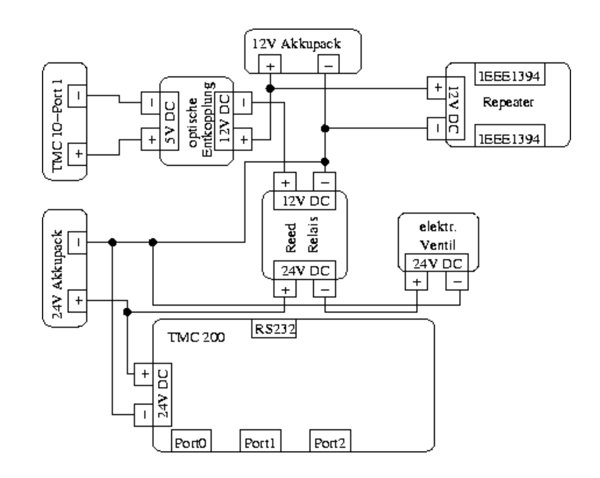


Abbildung B.5: Stromkabel zwischen den Roboterkomponenten (TMC)

für einen Motor läuft über zwei verdrillte Kupferdrähte statt der sechs Adern beim Kameleonboard, die über Schraubklemmen befestigt sind. Die Datenleitungen aller drei Encoder werden in einem 14-poligen Flachbandkabel zusammengefasst, aber es muss zusätzlicher Aufwand betrieben werden: ohmsche Widerstände sind zwischen einige der vier Datenleitungen pro Encoder einzulöten (siehe [6, 8]).

B.3 Ansteuerung Schusseinheit

Die Schusseinheit des Roboters wie sie in Abschnitt 2.4 beschrieben ist, benötigt einen Auslöser, genauer gesagt das eingesetzte elektrische Ventil (vgl. Abbildung 2.12) einen Impuls. In unserer Lösung ist der Motorcontroller für diesen Impuls verantwortlich, die spezifischen Ansätze sollen in den nächsten beiden Unterabschnitten näher erläutert werden.

B.3.1 Kameleon

Die Ansteuerung der Schusseinheit erfolgt auf recht einfache Weise: ein 5 V-Reed-Relais (Schließer mit Schutzdiode), dessen Eingangsseite zwischen +5 V und einem Digital Power Out-Ausgang gehängt wird (Pin 2 bzw. Pin25 in unserer Implementierung). Die Ausgangsseite des Relais wird einerseits mit +24 V verbunden und andererseits mit einem Pol des Ventils. Als Schaltbild ergibt sich Abbildung B.6.

Digitiale E/A-Ports des K376SBC können nicht genutzt werden, da sie maximal ca. 1mA Strom bereitstellen (vgl. [10], Seite 9), das Relais aber erst bei ca. 15mA auslöst. Das Schaltbild der gesamten Ansteuerung ist in Abbildung B.6 dargestellt.

B.3.2 TMC

Das TMC ermöglicht über insgesamt 10 E/A-Ports den Anschluss bzw. die Ansteuerung / Abfrage von Sensoren und Aktoren, wobei besondere Vorsicht geboten ist: zwischen den Pins der Ports und der angeschlossenen Elektronik ist auf saubere Signaltrennung zu achten. Am besten entkoppelt man beides optisch voneinander. Hierfür wird ein Optokoppler vom Typ PC817 eingesetzt, der über einen 270 Ohm Widerstand mit dem entsprechenden E/A-Pin des TMC verbunden wird. Der maximale Strom in diesem Kreis ist somit auf $\leq$20 mA begrenzt und liegt innerhalb der Spezifikationen des PC817.

Das elektrische Ventil kann nicht direkt an den Ausgang des Kopplers angeschlossen werden, dieser stellt nicht genug Strom zur Auslösung bereit. Ein zusätzlicher Querzweig mit einer Standarddiode

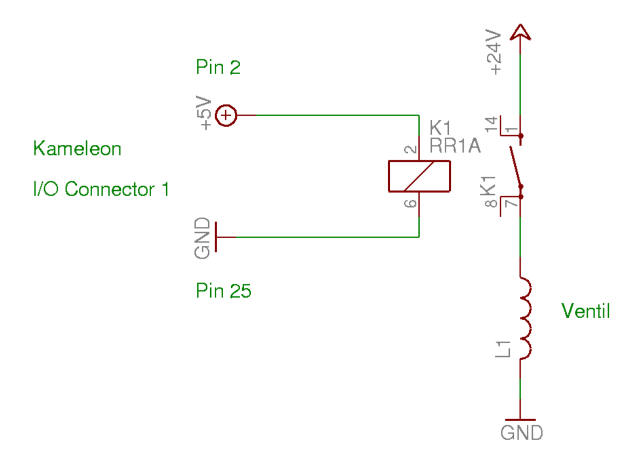


Abbildung B.6: Schaltplan für die Ansteuerung der Schusseinheit über Kameleon

vom Typ 1N148, einen Transistor BD139 und einer 12 V Spannungsquelle soll das Problem der Stromversorgung lösen.

Das Schaltbild der gesamten Ansteuerung ist in Abbildung B.7 dargestellt.

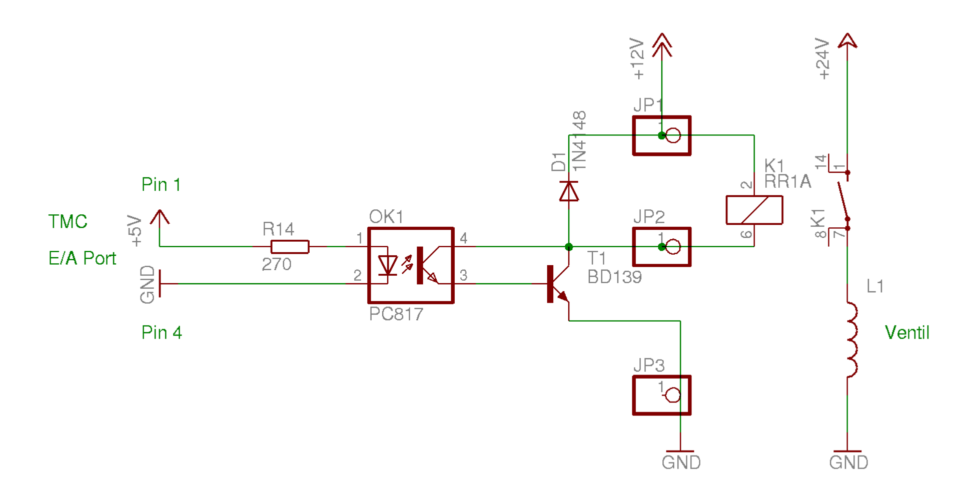


Abbildung B.7: Schaltplan für die Ansteuerung der Schusseinheit über TMC

B.4 Konfigurationsdateien

Das harte Codieren von Werten und Parametern macht bei einer Änderung eines Parameters oft das Neucompilieren des entsprechenden Programmabschnitts notwendig. Ein alternativer Ansatz ist die Auslagerung der relevanten Parameter in eine Konfigurationsdatei, die beim Programmstart geladen wird. In dem Projekt wurden zwei dieser Dateien eingesetzt, die im Folgenden näher beschrieben werden:

robotControl.cfg Diese Datei enthält Parameter für die Kommunikation zu den Motorcontrollern und für die Controller selbt.

- BOARD darf die Werte 1 und 2 annehmen und dient zur Unterscheidung, welcher Motorcontroller angeschlossen ist: Kameleon (1) oder TMC200 (2)
- USE legt fest, ob der Controller mit einem Rechner über PCMCIA (1) oder über eine normale serielle Schnittstelle verbunden ist
- COM_PORT verwendete Schnittstelle, z.B. 0
- COM_LAPTOP_STBITS Anzahl der Stopbits bei Kommunikation mittels der PCMCIA-Lösung
- COM_TOWER_STBITS Anzahl der Stopbits bei einer Kommunikation über eine normale serielle Schnittstelle
- ROB_WHEEL_RADIUS_DEF Raddurchmesser eines omnidirektionalen Rades [m]
- ROB_GEAR_PARAM_DEF Untersetzungsverhältnis des Motorgetriebes
- ROB_PULSES_PER_TURN_DEF Impulsanzahl des Encoders pro Umdrehung
- TMC_COM_SPEED: Kommunikationsgeschwindigkeit der seriellen Schnittstelle [Baud]
  Wertebereich $\{9600, 19200, 57600\}$
- TMC_WORKINGMODE: Arbeitsmodus des TMC200
  Wertebereich $[0; 2]$
- TMC_ANSWERMODE Antwortmodus des TMC200
  Wertebereich $[0; 5]$
- TMC_2ENCRES: Doppelte Encoderauflösung
  Wertebereich $[0; 5000]$
- TMC_MAXV: Maximale Motordrehzahl (Leerlaufdrehzahl $\left[\frac{\mathrm{U}}{\mathrm{min}}\right]$)
  Wertebereich $[0; 20000]$
- TMC_DELTADIST: Distanzänderung pro Encoderschritt $\left[\frac{\mu\mathrm{m}}{\frac{1}{2}\mathrm{Schritt}}\right]$
  Wertebereich $[0; 5000]$
- TMC_CMAX maximaler Anlaufstrom [10mA-Schritte]
  Wertebereich $[0; 1000]$
- TMC_CNOM Dauerlaststrom [10mA-Schritte]
  Wertebereich $[0; 1000]$
- TMC_MAX_TEMP maximal erlaubte Motorwicklungstemperatur [°C]
  Wertebereich $[0; 1000]$

- TMC_MIN_TEMP Abschalttemperatur der Strombegrenzung [°C]
  Wertebereich [0; 1000]
- TMC_P Proportional-Anteil des PID-Reglers
  Wertebereich [0; 1000]
- TMC_I Integral-Anteil des PID-Reglers
  Wertebereich [0; 1000]
- TMC_D Differential-Anteil des PID-Reglers
  Wertebereich [0; 1000]
- KAM_COM_SPEED Kommunikationsgeschwindigkeit der seriellen Schnittstelle
- KAM_P Proportional-Anteil des PID-Reglers
- KAM_I Integral-Anteil des PID-Reglers
- KAM_D Differential-Anteil des PID-Reglers

tribots.cfg

- CamDummy = [0,1] Kamera wird angesteuert / es werden Bilddaten aus einer Datei eingelesen
- CAM_FILENAME = Datei, die Bilddaten enthält (im YUV Format)
- TEST_MODE_ACTIV = [0,1] schaltet die Testroutinen an/aus
- TEST_TAKT = Dauer eines Schleifendurchlaufs aller Module in ms
- TEST_KICK = [0,1] schaltet Test der Kickeinheit an/aus
- TEST_ODO = [0,1] schaltet Test der Odometrie an/aus
- TEST_LINE_SL = [0,1] schaltet Selbstlokalisationstest auf einer Linie fahren an/aus
- TEST_LINE_SL_SPEED = Geschwindigkeit für den Test in m/s
- TEST_LINE_SL_DISTANCE = Distanz die gefahren wird in m
- TEST_SIDE_ACC = [0,1] schaltet den Test seitwaertsfahren an/aus
- TEST_SIDE_ACC_PASSES = Anzahl von seitwärtsbewegungen
- TEST_APPROACH_POINT = [0,1] schaltet den Test auf Punkt fahren an/aus
- TEST_APPROACH_POINT1_X = X-Koordinate des 1. Punktes
- TEST_APPROACH_POINT1_Y = Y-Koordinate des 1. Punktes
- TEST_APPROACH_POINT2_X = X-Koordinate des 2. Punktes
- TEST_APPROACH_POINT2_Y = X-Koordinate des 2. Punktes
- TEST_APPROACH_POINT3_X = X-Koordinate des 3. Punktes
- TEST_APPROACH_POINT3_Y = X-Koordinate des 3. Punktes
- TEST_APPROACH_POINT4_X = X-Koordinate des 4. Punktes

- TEST_APPROACH_POINT4_Y = X-Koordinate des 4. Punktes
- TEST_FOLLOW_HALSBAND = [0,1] schaltet den folge Halsband Test an/aus
- DISTANCE_CALIBRATION = [0,1] schaltet Disatnzkalibreirung an/aus, wird nicht benötigt da über GUI schaltbar
- MERGE_OBSTACLES = [0,1] schaltet zusammenfassen der Hindernisse an/aus
- MERGE_OBSTACLES_OFFSET = Bereich in mm in dem Hindernisse zusammengefasst werden
- TCN = Nummer des Roboters für TCPE
- TCPE = IP-Adresse des TCPE Rechners
- WRITE_IMAGE_TO_FILE = [0,1] schaltet abspeichern der Kamerabilder an/aus
- NUM_SAVED_IMAGES = Anzahl der Abzuspeichernden Bilder
- WRITE_IMAGES_WITH_SEGMENTATION = [0,1] gibt an ob die Bilder segmentiert abgespeichert werden
- SLIDE_IMAGES_AUTO = [0,1] Bilder werden automatisch durchgeblättert
- SLIDE_IMAGES_MANUAL = [0,1] Ein Bild wird solange als INput benutzt bis benutzer weiterschaltet
- STANDARD_PLAYERTYPE = Gibt den Initialisierungswert für den Spielertypen an

B.5 Kameleon - Kommunikationsprotokoll

B.5.1 Vorbemerkungen

In den folgenden Abschnitten steht Π für CR (carriage return) *oder* LF (line feed), sowie Γ für CR *und* LF.
Ein gesendeter Befehl hat immer das Format „*CharChar, Variable(i)*Π"mit $i \in \{0, 1, ..., n\}$, die dazugehörige Antwort hat das Format „*CharChar, Variable(i)*Γ", wiederum mit $i \in \{0, 1, ..., n\}$.

B.5.2 Liste der verfügbaren Kommandos

AA Konfiguration eines PID-Geschwindigkeitskontrollers
Befehl: AA,Motornummer,K_p,K_i,K_d Π
Antwort: aaΓ

AB Konfiguration aller PID-Geschwindigkeitskontroller
Befehl: AB,K_p,K_i,K_d Π
Antwort: abΓ

DA Setzen der Geschwindigkeit eines Motors
Befehl: DA,Motornummer,MotorgeschwindigkeitΠ
Antwort: daΓ
Einheit: Pulse/10ms

DB Setzen der Geschwindigkeiten aller *vier* Motoren
Befehl: DB,Motorgeschwindigkeit0,Motorgeschwindigkeit1,Motorgeschwindigkeit2, Motorgeschwindigkeit3Π
Antwort: dbΓ
Einheit: Pulse/10ms

EA Geschwindigkeit eines Motors auslesen
Befehl: EA,MotornummerΠ
Antwort: ea,MotorgeschwindigkeitΓ
Einheit: Pulse/10ms

EB Geschwindigkeit aller Motoren auslesen
Befehl: EBΠ
Antwort: eb,Motorgeschwindigkeit0,Motorgeschwindigkeit1, Motorgeschwindigkeit2, Motorgeschwindigkeit3Γ
Einheit: Pulse/10ms

FB Setzen bzw. Widerrufen der Erlaubnis zum Kicken
Befehl: FB,xΠ mit $x \in \{0, 1\}$
Antwort: fbΓ
Standard: Freigabe ist deaktiviert.

GA Positionszähler eines Motors setzen
Befehl: GA,Motornummer,MotorpositionΠ
Antwort: gaΓ
Einheit: Pulse oder A/D-Bit.

GB Positionszähler aller Motoren setzen
Befehl: GB,Motorposition0,Motorposition1,Motorposition2,Motorposition3Π
Antwort: gbΓ
Einheit: Pulse oder A/D-Bit.

HA Positionszähler eines Motors auslesen
Befehl: HA,MotornummerΠ
Antwort: ha,MotorpositionΓ
Einheit: Pulse oder A/D-Bit.

HB Positionszähler aller Motoren auslesen
Befehl: HBΠ
Antwort: hb,Motorposition0,Motorposition1,Motorposition2,Motorposition3Γ
Einheit: Pulse oder A/D-Bit.

IA Auslesen der A/D Eingänge
Befehl: IA,KanalnummerΠ
Antwort: ia,analoger WertΓ
Einheit: 20mV/Bit bzw. 2.79mA/Bit

JB Auslesen aller Motorströme
Befehl: JBΠ
Antwort: jb,Strom0,Strom1,Strom2,Strom3Γ
Einheit: 2.79mA/Bit

NB Stoppen aller Motoren
Befehl: NBΠ
Antwort: nbΓ

PA PWM-Amplitude eines Motors setzen
Befehl: PA,Motornummer,MotorPWMΠ
Antwort: paΓ
Wertebereich: -100% bis 100% entspricht -255 bis 255

PB PWM-Amplitude aller Motoren setzen
Befehl: PB,Motor0PWM,Motor1PWM,Motor2PWM,Motor3PWMΠ
Antwort: pbΓ
Wertebereich: -100% bis 100% entspricht -255 bis 255

QA Starten des LPRSerCom-Protokolls
Befehl: QAΠ
Antwort: qaΓ

QB Starten des SerCom-Protokolls
Befehl: QBΠ
Antwort: qbΓ

SB Abfragen aller möglichen Sensordaten
Befehl: SBΠ
Antwort: sb,SPD0,SPD1,SPD2,SPD3,Voltage,CUR0,CUR1,CUR2,CUR3,
SEN0,SEN1,SEN2,SEN3,SEN4,SEN5Γ

SPDx - akt. Geschwindigkeit, Einheit: pulse/10ms
Voltage - Spannung der Batterie [mV]
CURx - Strom am Motor x [0-3] [mA]
SENx - Ultraschallsensor x [0-5] [cm]

UB Ultraschallmeßwerte abfragen
Befehl: UBΠ

Antwort: ub,sensor0,sensor1,sensor2,sensor3,sensor4,sensor5Γ
Einheit: cm

VB Version des Protokolls auslesen
Befehl: VBΠ
Antwort: vb,ProtokollversionΓ

ZB Angeschlossene Motoren feststellen
Befehl: ZBΠ
Antwort: zb,motor0,motor1,motor2,motor3Γ
0 - Motor nicht angeschlossen
1 - Motor angeschlossen

Literaturverzeichnis

[1] A. Bonarini, G. Kraetzschmar, P. Lima, T. Nakamura, F. Ribeiro, and T. Schmitt. Middle size robot league - rules and regulastions for 2003, März 2003.

[2] J.R. Carpenter, P. Clifford, and P. Fearnhead. Sampling strategies for monte carlo filters of non-linear systems., 2000.

[3] Arnaud Doucet. On sequential monte carlo sampling methods for bayesian filtering, 1998.

[4] S. P. Engelson and D. V. McDermott. Error correction in mobile robot map learning. In *IEEE International Conference on Robotics and Automation*, pages 2555–2560, Nizza, Frankreich, Mai 1992.

[5] Roland Hafner. Reinforcement Lernen auf mobilen Robotern. Diplomarbeit, Universität Karlsruhe, 2002.

[6] Fraunhofer Institut. *TMC Handbuch*. Bonn, 2003.

[7] Sascha Lange. Tracking of color-marked objects in a 2-dimensional space. Bachelor-Arbeit, Universität Osnabrück, 2001.

[8] maxon motor AG. *Quick Assembly Two and Three Channel Optical Encoders*, April 2000.

[9] K-TEAM S.A. *Kameleon 376SBC User Manual*. Lausanne, 1999.

[10] K-TEAM S.A. *Kameleon Robotics Extension User Manual*. Lausanne, 2001.

[11] Sony. *Technical Manual Ver1.0 -English-*, 2001.

[12] S. Thrun, D. Fox, W. Burgard, and F. Dellaert. Robust monte carlo localization for mobile robots. *Artificial Intelligence*, 128(1-2):99–141, 2000.